Symbolic Analysis of Cryptographic Protocols

Models, Methods, and Soundness

PhD Thesis - Morten Dahl Jørgensen
Supervisor - Ivan Damgård
March 2013

Department of Computer Science
Science and Technology
Aarhus University

Acknowledgements

First of all I would like to thank Ivan Damgård for welcoming me, being an active advisor, and always keeping his door open and taking the time to answer questions.

I would like to thank my hosts and co-authors through-out the years – Hans Hüttel, Graham Steel, Ran Canetti, Stéphanie Delaune, Tomas Toft, Naoki Kobayashi, Yunde Sun, and Chao Ning – for their hospitality, time, and collaboration, and the numerous people with whom I have had many fruitful discussions.

Finally, a warm thank you to my friends and family, especially to my girlfriend and the three from Vestergade, for their endless encouragement, support, and patience.

Morten Dahl Jørgensen
Aarhus, March 2013

Preface

This thesis is based upon two published papers and an unpublished technical report:

- *Type-based Automated Verification of Authenticity in Asymmetric Cryptographic Protocols* was done primarily at Aalborg University in collaboration with Naoki Kobayashi, Yunde Sun, and Hans Hüttel. It was published at *The 9th International Symposium on Automated Technology for Verification and Analysis 2011* (ATVA'11).
- *Formal Analysis of Privacy for Anonymous Location Based Services* was done at École Normale Supérieure de Cachan in collaboration with Graham Steel and Stéphanie Delaune. It was published at *Theory of Security and Applications 2011* (TOSCA'11).
- *Universally Composable Symbolic Analysis for Two-Party Protocols based on Homomorphic Encryption* was done at Aarhus University in collaboration with Ivan Damgård. Several of the initial ideas emerged from fruitful discussions with Ran Canetti while visiting Boston University.

Two other papers were also published but have been omitted in the name of coherency:

- *Formal Analysis of Privacy for Vehicular Mix-zones* was done at École Normale Supérieure de Cachan in collaboration with Graham Steel and Stéphanie Delaune. It was published at *The 15th European Symposium on Research in Computer Security 2010* (ESORICS'10).
- *On Secure Two-party Integer Division* was done at Aarhus University in collaboration with Tomas Toft and Chao Ning. It was published at *The 16th Conference on Financial Cryptography and Data Security 2012* (FC'12).

English Résumé

We present our work on using abstract models for formally analysing cryptographic protocols:

- First, we present an efficient method for verifying trace-based authenticity properties of protocols using nonces, symmetric encryption, and asymmetric encryption. The method is based on a type system of Gordon et al., which we modify to support fully-automated type inference. Tests conducted via an implementation of our algorithm found it to be very efficient.
- Second, we show how privacy may be captured in a symbolic model using an equivalence-based property and give a formal definition. We formalise a concrete protocol by Blumberg et al. using one-way functions and commitments for providing various location based vehicle services, and report on our findings and experience of carrying out its analysis using the ProVerif tool.
- Third, we make an abstract model of the powerful simulation-based *Universally Composable* framework by Canetti for specifying and analysing protocols, and show that our model is sound with respect to its standard computational interpretation. Our model supports powerful primitives such as homomorphic encryption and non-interactive zero-knowledge proofs, which we show may be used to implement several interesting two-party functionalities. As a case study we use the ProVerif tool to analyse an oblivious transfer protocol by Damgård et al. under static corruption.

Dansk Resumé

Vi præsenterer vores arbejde vedrørende brugen af abstrakte modeller til formel analyse af kryptografiske protokoller:

- Først præsenterer vi en effektiv metode til at verificere autentifikationsegenskaber for protokoller, der gør brug af symmetrisk og asymmetrisk kryptering. Metoden bygger på et typesystem af Gordon m.fl., som vi modificerer således, at typeinferens kan lade sig gøre. Testresultater fra en implementation indikerer dernæst, at vores algoritme er effektiv.
- Efterfølgende viser vi, hvordan anonymitet kan udtrykkes symbolskt ved hjælp af en ækvivalens. Vi formaliserer en konkret protokol af Blumberg m.fl., der via en-vejs-funktioner og commitment schemes understøtter lokationsbaseret tjenester for køretøjer. Vi dokumenterer resultatet af dens analyse samt erfaringer fra brugen af ProVerif-værktøjet.
- Til sidst giver vi en abstrakt model af Canettis simulationsbaseret *Universally Composable* platform til specifikation og analyse af protokoller og viser sundhed i forhold til standardfortolkningen. Vores model understøtter homomorfisk kryptering og zero-knowledge-beviser, der tilsammen tillader os at implementere flere interessante toparts-funktionaliteter. Vi tester anvendeligheden af modellen ved at bruge ProVerif-værktøjet til at analysere en oblivious transfer-protokol af Damgård m.fl. under statisk korruption.

Table of Contents

Chapter 1

Introduction

A *cryptographic protocol* describes how a set of players with access to cryptographic primitives should behave in order to collaboratively perform a certain computation. In this thesis we focus on how such protocols may be analysed, that is, how we may gain confidence that they behave as intended even under the attack of an arbitrary adversary trying to break them.

As a first step we need to define a protocol model in which we can capture the players, their primitives, and adversaries. Since protocols are executed by computers it may be argued that the *computational models* are the most realistic and give the highest level of certainty. However, the fact that these models include many technical details may complicate matters to the point where tool supported analysis becomes infeasible, and where even manual analysis is done somewhat informally and hence prone to error.

One approach for dealing with the above complexity issue is to instead consider abstract models ignoring certain details that are not deemed important for the analysis. In this thesis we focus on a particular kind of abstract models, namely those allowing for a *symbolical analysis*, and argue that they may yield a good balance between ease of use and realistic security guarantees, by not only opening up for fully-automated formal analysis, but also being powerful enough to express advanced protocols and their security properties. Moreover, we also show how they can be proved sound with respect to well-established computational models and as a result provide a high degree of certainty.

In this chapter we first sketch the particular kind of symbolic models we will be using[1] and then we outline the content of the rest of the thesis: in Section 1.2 (based on Chapter 2) we present an efficient method for automating the analysis of trace-based authenticity properties using type systems; in Section 1.3 (based on Chapter 3) we perform a concrete analysis of a protocol, which includes investigating how the more complex equivalence-based security properties such as privacy may be captured symbolically, and how current tools can be used to automate their verification; finally, in Section 1.4 (based on Chapter 4) we give a computational sound model for analysing advanced primitives and equivalence-based security properties in the simulation-based paradigm, and conduct a concrete analysis as a case study in the expressive power of the model and to what extend it allows for automation.

Related work is given in each chapter with respect to its aspect of symbolic analysis.

1.1 Symbolic Models

The symbolic models used here differ from the computational models by abstracting away things such as the security parameter and the negligible probability with which the adversary may

[1] Several symbolic models exist in the literature, including the quintessential Dolev-Yao [DY83] and BAN logic [BAN90]. We have focused on the process calculi here due to their wide-spread use and tool support.

break a protocol by for instance guessing a secret key: with a notion of adversarial knowledge and unguessable atomic symbols, dubbed *names*, is it possible to specify logical rules saying that the only manipulation an adversary can do to a ciphertext is to compare it to other messages and obtain the plaintext if he knows the (unguessable) name corresponding to the decryption key. On the other hand, in the computational models no prior information may still allow the adversary to obtain the length of an encrypted plaintext, knowing most of a decryption key may allow him to guess the rest, and knowing the decryption key may reveal the randomness used for a ciphertext. The obvious advantage of the computational models is that they have a tight connection with the real execution of a protocol and hence give a high degree of accuracy, including the certainty of a security analysis. However, it comes with the price that tool support is sparse and both manual and automated analysis quickly becomes an expert task or done informally.

The Spi-Calculus

Milner et al. [MPW92, Mil99] introduced the *π-calculus* that uses *processes* to model communicating systems passing around atomic data packages. It has been a highly successful model for various purposes (see for example [VM94, PT97, PW05, PC07]), including the idea of analysing cryptographic protocols by for instance comparing a protocol to its ideal behaviour (or specification) through an equivalence relation between the two systems. However, while it is capable of expressing cryptographic protocols, it has to do so through an encoding that overcomplicates matters and shields away many of the assumptions put on the primitives.

As an attempt to remedy this, Abadí and Gordon introduced the *spi-calculus* [AG99] as an extension of the π-calculus with explicit *terms* for modelling cryptographic messages such as ciphertexts, and encryption and decryption processes that explicit model the behaviour of these procedures by for instance making it clear when decryption succeeds and when it fails. They also developed several analysis methods for their model, including carrying over the idea of specifying security properties such as secrecy and authenticity through equivalences. There have since been an array of variants of the calculus extending it with other primitives, as well as work on methods and tools, not least in the form of type systems for trace-based security properties [Aba99, GJ04, HJ04, FGM07]. In Section 1.2 we sketch a spi-calculus with symmetric and asymmetric encryption, and a type system for proving authenticity properties.

The Applied-Pi Calculus

The ad-hoc development of variants of the spi-calculus has its obvious limitation in that each variant must give an operational semantics, or even provide fundamental theorems[2]. In an effort to unify the many spi-calculi, Abadí and Fournet introduced the *applied-pi-calculus* [AF01] that is parameterised by term constructors and an equivalence theory giving the semantics for these. They give a general semantics as well as observational equivalence and its characterising bisimularity. This calculus has seen wide-spread acceptance in recent years, due not least to its ability to model analyse complex systems such electronic voting schemes [DKR09] and advanced primitives such as zero-knowledge proofs [BMU08], but also to the existence of tool support such as ProVerif [BACS13] that may handle both trace-based and equivalence-based security properties. In Section 1.3 we instantiate the applied-pi calculus with term constructors for one-way functions and commitments, and in Section 1.4 with term constructors for group arithmetic, commitment, homomorphic encryption, and non-interactive zero-knowledge proofs.

[2]For instance, when a calculus is analysed using equivalence-based security properties one might be interested in providing results making observational equivalence, which quantifies over all contexts, easier to analyse; this is often done via the notion of *bisimilarity* which removes quantification in exchange for step-wise equivalence.

1.2 Automated Analysis of Authenticity

In Chapter 2 we present our work on automating the analysis of trace-based authenticity properties in symbolic models. We use a variant of the spi-calculus with symmetric and asymmetric encryption as our protocol model and for embedding security properties. By using a type system to prove that the properties are satisfied we may obtain a highly efficient and automated way of proving authenticity through a type inference algorithm.

Symbolic analysis using type systems has been an active area of research and by now includes type systems for several trace-based security properties. The typical advantages are that protocols can be verified in a modular manner, and that an explicit and easily verifiable proof is provided in the form of the types. Furthermore, it is typically relatively easy to extend the approach to verify actual source code instead of models. Their disadvantages however, include that users must provide complex type annotations that require expertise in both security protocols and type theories; by giving a type inference algorithm we remove this task and automate the use of type systems.

Protocol Model

The *messages*, ranged over by M, are given by the terms inductively defined by[3]:

$$x \mid \mathbf{pair}(M_1, M_2) \mid \mathbf{senc}(M, k) \mid \mathbf{aenc}(M, k)$$

where x, k are *names*, $\mathbf{pair}(M_1, M_2)$ is a concatenation of M_1 and M_2, and $\mathbf{senc}(M, k)$ and $\mathbf{aenc}(M, k)$ represents the ciphertext[4] obtained by encrypting M with respectively symmetric and asymmetric key k. For asymmetric encryption we do not distinguish between encryption and signing, hence $\mathbf{aenc}(M, k)$ denotes an encryption if k is a public key and a signing if k is a private key. Protocols are expressed as *processes*, ranged over by P, that allows for input/output, nonce generation and equality checking, symmetric and asymmetric key generation, and message manipulation such as encryption and decryption:

$$\begin{gathered}\mathbf{nil} \mid \mathbf{out}[c, M] \mid \mathbf{in}[c, x]; P \mid (P_1 \parallel P_2) \mid !\, P \mid \mathbf{new}\, x; P \mid \mathbf{new}_s\, x; P \mid \mathbf{new}_a\, x, y; P \\ \mid \mathbf{check}\ M_1\ \mathbf{is}\ M_2; P \mid \mathbf{split}\ M\ \mathbf{is\ pair}(x, y); P \mid \mathbf{match}\ M_1\ \mathbf{is\ pair}(M_2, y); P \\ \mid \mathbf{decrypt}\ M\ \mathbf{is\ senc}(x, k); P \mid \mathbf{decrypt}\ M\ \mathbf{is\ aenc}(x, k); P\end{gathered}$$

where, as a few examples: process **nil** does nothing; process $\mathbf{out}[c, M]$ sends M over name c, and $\mathbf{in}[c, x]; P$ waits for some message M on name c and then behaves as P with M substituted for x; process $P_1 \parallel P_2$ executes P_1 and P_2 in parallel, and $!\, P$ executes infinitely many copies of P in parallel; process $\mathbf{new}\, x; P$ generates a fresh ordinary name (such as atomic messages and nonces), $\mathbf{new}_s\, x; P$ generates a fresh symmetric key, and $\mathbf{new}_a\, x, y; P$ generates a fresh asymmetric key pair.

Security Properties

Authenticity properties are modelled through *correspondence assertions* [WL93], and expressed directly in the calculus by adding two processes:

$$P ::= \ \ldots \mid \mathbf{begin}\ M; P \mid \mathbf{end}\ M$$

[3] For consistency with the rest of the chapter we use a slightly different syntax here than in Chapter 2.

[4] Note that we do not model the randomness that may be used when forming ciphertexts in the real world. This is justified by the fact that if a plaintext contains at least one nonce (modelled by the unguessable names) then the entire plaintext is unguessable.

that do not change the operation semantics but simply raises events: process **begin** $M; P$ raise a begin event for M and then behaves like P, while **end** M just raises an event event for M.

We intuitively call a process *safe* (with respect to its embedded correspondence assertions) if for each end-event occurring, a corresponding begin-event has already occurred, and *robustly safe* if a process is safe in the presence of arbitrary attackers representable as processes in our calculus. As in most verification methods we aim at proving robust safety automatically.

To formalise the safety notions we give an operational semantics via runtime states, where the special runtime state `Error` is entered when the correspondence assertions have been violated. We say that a process O is an *adversary* (also called *opponent*) if it contains no begin, end, nor check operations. Using the operational semantics, robust safety is defined as follows:

Definition 1.2.1 (Safety and Robust Safety)**.** *A process P is* safe *if there is no reduction from its initial state to* `Error`*. A process P is* robustly safe *if $P \mid\mid O$ is safe for every adversary O.*

Type System

We next give a type system as a proof technique for proving that well-typed processes are robustly safe. This allows us to reduce protocol verification to type checking, and in turn to automate verification via type inference. We use the notion of *capabilities* in order to statically guarantee that end-events can be raised only after the corresponding begin-events. A capability φ is a multiset of *atomic capabilities* of the form $\mathsf{end}(M)$ expressing a permission to raise an end event.

The robust safety of processes is guaranteed by enforcing the following conditions on capabilities: (i) to raise an **end** M event, a process must possess and consume an atomic $\mathsf{end}(M)$ capability; and (ii) an atomic $\mathsf{end}(M)$ capability is generated only by raising a **begin** M event. These conditions can be statically enforced by using a type judgment of the form $\Gamma; \varphi \vdash P$, which means that P can be safely executed under the type environment Γ and the capabilities described by φ. For example, for $\Gamma = x : T$ for some type T, judgment $\Gamma; \{\mathsf{end}(x)\} \vdash$ **end** x is valid but $\Gamma; \emptyset \vdash$ **end** x is not. This can be locally enforced by the following typing rules for begin and end processes:

$$\frac{\Gamma; \varphi + \{\mathsf{end}(M)\} \vdash P}{\Gamma; \varphi \vdash \mathbf{begin}\ M; P} \qquad\qquad \frac{}{\Gamma; \varphi + \{\mathsf{end}(M)\} \vdash \mathbf{end}\ M}$$

where the left rule for begin makes the new capability $\mathsf{end}(M)$ available after the begin-event, and the right rule for end ensures that the capability is available.

The main difficulty lies in how to pass capabilities between processes. We do this by attaching capabilities to nonce names, and introduce types of the form $\mathbf{N}(\varphi_1, \varphi_2)$ which describes names carrying an *obligation* φ_1 that must be satisfied, and a capability φ_2 can may be used. This is similar to what is done in [GJ04], however our uniform treatment not only allows us to reduce type inference to a problem of solving constraints on capabilities and obligations, but also allows us to express a wider range of protocols, such as cryptographic protocols with several players. Formally, for $\mathbf{Un} \doteq \mathbf{N}(\emptyset, \emptyset)$ we obtain the following result for our type system:

Theorem 1.2.2 (Soundness)**.** *Let P be any process with free names $x_1, \ldots, x_n$. If judgment $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ is valid in the type system then P is robustly safe.*

Type Inference

The final step of our approach is to automate the process of proving type judgments. In other words, given as input a process P with free names $x_1, \ldots, x_n$, we want to not only decide if a proof tree for judgment $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ exists, but also explicitly construct it.

Our algorithm is an extension of [KK09] to our more powerful type system: after determining the basic shape of types it generates a set of constraints on the capabilities that can then be reduced to linear programming.

1.3 Capturing and Analysing Privacy

In Chapter 3 we perform a concrete symbolic analysis of the VPriv scheme of Blumberg et al. [BBP09]. But besides expressing and analysing the protocol formally, we also investigate how privacy may be captured in the symbolic model; in light of the previous subsection we here consider a equivalence-based security property that is not determined by the behaviour of a single process but instead as a relationship between two processes. This makes the property harder to check, with the consequence that focus here is moved from constructing efficient fully-automated methods, to the question of how to define properties such as privacy, how to do so in a way suitable for symbolic models, and how to work with them in current tools.

The VPriv Scheme

The VPriv scheme offers a variety of location-based vehicular services such as "pay-as-you-go" insurance and electronic toll collection. The participants are a single service provider and a set of vehicles, and the goal of the scheme is to both protect the privacy of drivers whilst ensuring that they cannot cheat the service provider by, for instance, paying a lower price.

We assume that time is split into periods, say of length one month. The following three phases are then executed in order by each vehicle during each period. At the start of a period (*registration phase*), the vehicle generates fresh random tags for the period and registers commitments to hashed versions of these with the service provider. Then (*driving phase*), whenever the vehicle must emit a message containing an identifier during the period it will choose a new tag from its set of fresh tags. The tags are emitted in clear and the service provider records all tags v emitted by all vehicles together with the emission location l and a timestamp t, building a database containing a mixture of tuples (v, t, l). Finally, at the end of a period (*reconciliation phase*), each vehicle initiates a secure function evaluation protocol with the service provider in order to compute and settle the payable debts.

The informal privacy definition stated in [BBP09] asks that the privacy guarantees from the system are the same as those of a system in which the server, instead of storing tuples (v, t, l), stores only tag-free path points (t, l). In other words, from the server's point of view, the tags might just as well be uncorrelated and random. This definition accounts for the fact that some privacy leaks are unavoidable and should not be blamed on the system. For instance, if one somehow learns that only a single vehicle was on a certain road at a particular time, then that vehicle's tags can of course be linked to the tags emitted along the road at that time.

Protocol Model

To symbolically model the protocol we instantiate the applied-pi-calculus with term constructor $\mathbf{f}(x)$ to model the one-way function and $\mathbf{commit}(x, ck)$ to model commitments. We do not give a way to invert term $\mathbf{f}(x)$ to obtain x, but we have term constructor **open** that by rule

$$\mathbf{open}(\mathbf{commit}(x, ck), ck) = x$$

intuitively allows one to open (invert) a commitment if the correct key ck is known. Using this model we may give processes for all honest parties involved, in particular the vehicles and the service provider.

Definition of Privacy

Intuitively, our definition of privacy says that the service provider is unable to determine which route a vehicle took. For simplicity we assume that the vehicle has a choice of two routes, $route_{left}$ or $route_{right}$, and the property hence becomes whether or not the process corresponding to the vehicle taking $route_{left}$ is indistinguishability from the same process taking $route_{right}$ instead. This equivalence-based property seems somewhat natural, while on the other hand it is less clear how to capture this strong notion of privacy as a trace-based property as we did for authenticity above (which was a question of a single process reaching a bad state).

As done in previous symbolic modelling of similar properties, one might a priori be tempted to use observational equivalence as the notion of indistinguishability. Yet as it happens, this notion turns out to be too strong for in our analysis of the VPriv scheme, and as a result we instead rely on *trace equivalence*. Formally, let a process V_A be given that models the behaviour of vehicle A in a scheme such as the VPriv scheme. For simplicity we assume that it emits a single tag along its route. However, to account for the fact that it is trivial to deduce where the vehicle went if it is the only one, we also assume a process V_B^{dri} that emits a tag at the route not visited by vehicle A as a "counter-weight" yet does otherwise not interact with the service provider; in the case of the VPriv scheme it hence only performs the driving phase. We then say that a scheme ensures *privacy* if the following trace equivalence holds:

$$C_T\big[\, V_A(route_{left}) \mid V_B^{\mathsf{dri}}(route_{right})\big] \sim_t C_T\big[\, V_A(route_{right}) \mid V_B^{\mathsf{dri}}(route_{left})\big]$$

where C_T is an evaluation context modelling additional assumptions that may have to be made for the property to hold, such as the server being semi-honest (curious but following the protocol), or the existence of a trusted third party helping vehicles perform sanity checks on the list of tags received from the server during the reconciliation phase.

Evaluation

Through our analysis using the ProVerif tool we identity specific circumstances in which privacy can be violated in the VPriv scheme, including cases where the vehicles fail to perform sanity checks on the list received from the server. Based on these findings we suggest fixes that then allow us to prove privacy.

The work also presents a concrete situation in which observational equivalence is too strong and trace equivalence must be used instead. This is unfortunate not only because tool support exists for observational equivalence, but also because it is a congruence and hence useful for compositional analysis. Intuitively, the problem is that step-wise simulation is required by observational equivalence whereas trace equivalence allows us to simulate only after the execution is complete. This problem is well-known in the computational setting and hints that if we want to use a step-wise equivalence in the symbolic setting then one solution might be to adapt some of the tricks use there. This is made clear in our following work where *extraction trapdoors* play an important role.

1.4 Computational Sound Composable Analysis

In Chapter 4 we turn to another approach for specifying security properties, namely the simulation-based paradigm of the computational *Universally Composable (UC)* framework by Canetti [Can01]. Here, *ideal functionalities* expressible within our protocol model are used to specify the expected behaviour of protocols, with the consequence that the security properties we consider are not only not a priorly fixed as they were above, we may also capture

requirements which it is less clear how to do using the previous approaches. We consider more advanced protocols and primitives as well, yet show through a case study that it is still possible to benefit from tools support. Finally, we address the question of realism of our symbolic model by providing a computational soundness result.

We focus on two-party function evaluation protocols and the primitives used by many of these, namely homomorphic public-key encryption, commitments, and certain zero-knowledge proofs. We consider an active adversary that may corrupt one of the players initially.

Protocol Model

Since we need to relate a symbolic and a computational model for the soundness result, we introduce a simple high-level programming language for specifying a system of players, ideal functionalities, and *simulators* (see below), and provide compilations to both models: for the symbolic model we produce a set of processes in an instantiation of the applied-pi-calculus, and for the computational model a set of ITMs fitting with the UC framework.

The language allows an entity to use input, output, conditionals, and invocation of operations; which operations are available depends on the kind of the entity and the corruption scenario. Ideal functionalities, for a start, may use operations:

$$\mathsf{isValue}(x) \to b, \mathsf{eqValue}(v_1, v_2) \to b, \mathsf{inType}_U(v) \to b, \mathsf{inType}_T(v) \to b, \mathsf{peval}_f(v_1, v_2, v_3, v_4) \to v,$$
$$\mathsf{isConst}(x) \to b, \mathsf{eqConst}_c(v) \to b, \mathsf{isPair}(x) \to b, \mathsf{pair}(x_1, x_2) \to x, \mathsf{first}(x) \to x_1, \mathsf{second}(x) \to x_2$$

for processing values, constants, and pairings of these. Players may additionally use:

$$\mathsf{isComPack}(x) \to b, \mathsf{isEncPack}(x) \to b, \mathsf{isEvalPack}(x) \to b, \mathsf{decrypt}_{dk}(c) \to v,$$
$$\mathsf{commit}_{U,ck,crs}(v, r) \to d, \mathsf{encrypt}_{T,ek,crs}(v, r) \to c, \mathsf{eval}_{e,ek,ck,crs}(c_1, c_2, v_1, r_1, v_2, r_2) \to c,$$
$$\mathsf{verComPack}_{U,ck,crs}(d) \to b, \mathsf{verEncPack}_{T,ek,crs}(c) \to b, \mathsf{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2, [d_1, d_2]) \to b$$

to generate commitments and encryptions, decrypt ciphertexts under their own encryption key, and verify proofs from the other player: when using operations commit_U and $\mathsf{encrypt}_T$ a non-interactive zero-knowledge proof is also generated, proving that the plaintext is in set T and U, respectively, and when using operation eval_e a proof that the ciphertext c was formed as the result of an homomorphic evaluation of expression e on the inputs. Finally, again in addition to the plain operations above, a simulator may also use operations:

$$\mathsf{isComPack}(x) \to b, \mathsf{isEncPack}(x) \to b, \mathsf{isEvalPack}(x) \to b,$$
$$\mathsf{simcommit}_{U,ck,simtd}(v, r) \to d, \mathsf{simencrypt}_{T,ek,simtd}(v, r) \to c,$$
$$\mathsf{simeval}_{e,ek,ck,simtd}(c_1, c_2, v_1, r_1, v_2, r_2) \to c, \mathsf{simeval}_{e,ek,ck,simtd}(v, c_1, c_2, d_1, d_2) \to c$$

to simulate the behaviour of an honest player, and operations:

$$\mathsf{extractCom}_{extd}(d) \to v, \mathsf{extractEnc}_{extd}(c) \to v, \mathsf{extractEval}_{1,extd}(c) \to v, \mathsf{extractEval}_{2,extd}(c) \to v$$

to extract values from the commitments and encryptions of a corrupt player.

As the simulation-based paradigm is fundamentally based on indistinguishability, we also need a notion of equivalence of systems in both models, namely the usual computational notion for all polynomial time adversaries in the computational model, and observational equivalence in the symbolic model.

Security Properties

In this work, the security of a protocol ϕ is defined relative to its expected behaviour in the form for an ideal functionality $\mathcal{F}$. More specifically, we say that ϕ is secure with respect to

$\mathcal{F}$ if no adversary can tell the difference between interacting with ϕ and interacting with $\mathcal{F}$ running together with some simulator *Sim*. We also say that the protocol *realises* the ideal functionality when we do not care about the simulator beyond the fact that it exists.

In light of the our previous work above, specifying the security requirements through ideal functionalities allows us to naturally capture the security guarantees under corruption of several protocols. For instance, while the requirements for an oblivious transfer protocol have a natural characterisation as an ideal functionality, it is less clear how to capture these using either of the two previous approaches.

Compositional Analysis

Another benefit of adapting the paradigms of the UC framework is that we may decompose a protocol into sub-protocols that are analysed independently. This reduces the complexity of the analysis and aids both manual and automated efforts. More specifically, a protocol Φ may be analysed with respect to ideal functionality $\mathcal{F}$ as follows:

1. decompose Φ into sub-protocols $\Phi_1, \ldots, \Phi_n$ and protocol ϕ using these, i.e. $\Phi = \phi^{\Phi_1,\ldots,\Phi_n}$
2. formulate ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_n$ and show that Φ_i realises $\mathcal{F}_i$
3. show that $\phi^{\mathcal{F}_1,\ldots,\mathcal{F}_n}$, i.e. the protocol using $\mathcal{F}_i$ instead of Φ_i, realises $\mathcal{F}$
4. use a general composition result to conclude that Φ also realises $\mathcal{F}$

where the analysis of Φ is broken into several analyses that may be done independently and in different models, i.e. in our symbolic model or directly in the computational model: since a prerequisite for doing an analysis in our model is that the (sub-)protocol, its ideal functionalities, and the corresponding simulator are expressible in our high-level language, decomposition may in some cases allow one to remove the parts of a protocol that our language cannot handle and then simply do them by hand.

In terms of reducing the complexity of an analysis, the point is that an ideal functionality is often simpler than the protocol realising it: intuitively, the protocol is split into the ideal functionality containing the core behaviour, and the simulator containing the unimportant "fluff" caused by e.g. having to use cryptographic primitives that in a sense may be cut away.

Computational Soundness

Our aim is to show that if we are given a proof in the symbolic model that a protocol ϕ realises an ideal functionality $\mathcal{F}$ then it follows that ϕ also realises $\mathcal{F}$ in the computational model. We actually show something slightly stronger, namely that (for our protocol class) observational equivalence in the symbolic model implies indistinguishability in the computational model.

As a first step we introduce a third intermediate model, which also produces a set of ITMs fitting into the computational UC framework, but which use a global ideal "crypto-box" that receives all calls to cryptographic operations and returns handles to objects such as encrypted plaintexts while storing these plaintexts in its restricted memory. Players then send such handles instead of actual ciphertexts and commitments, and the adversary is given a restricted interface to the memory through which he is effectively forced to launch his attack.

We then prove two soundness theorems stating that observational equivalence between two systems in the symbolic model implies computational indistinguishability of the two systems in the intermediate model, and computational indistinguishability in the intermediate model implies computational indistinguishability in the computational model.

With this result we then arrive at the following: to prove usual UC security of a protocol in our class it is hence sufficient to show observational equivalence of the compiled processes in

the symbolic model[5], which may (in part) be done using automated tools.

1.4.1 Case Studies

To investigate the expressibility of the approach and our high-level programming language, we show that several interesting protocols, including coin-flipping and multiplication-trip generation, may be captured. As a case study we furthermore analyse an oblivious transfer protocol and show how the ProVerif tool may be used to analyse and prove it secure.

1.5 Bibliography

[Aba99] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 104–115, New York, NY, USA, 2001. ACM.

[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.

[BACS13] Bruno Blanchet, Xavier Allamigeon, Vincent Cheval, and Ben Smyth. Proverif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/, 2013.

[BAN90] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions On Computer Systems*, 8:18–36, 1990.

[BBP09] Andrew J. Blumberg, Hari Balakrishnan, and Raluca Popa. VPriv: Protecting privacy in location-based vehicular services. In *Proc. 18th Usenix Security Symposium*, 2009.

[BMU08] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *IEEE Symposium on Security and Privacy*, pages 202–215, 2008.

[Can01] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings of 42nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*, pages 136–145, 2001.

[DKR09] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, July 2009.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[FGM07] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[5] Suitable ideal functionalities and simulators must also be constructed as part of the analysis. We do not deal with automating this, sometimes straight-forward, task here.

[GJ04] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

[HJ04] Christian Haack and Alan Jeffrey. Cryptyc. http://www.cryptyc.org/, 2004.

[KK09] Daisuke Kikuchi and Naoki Kobayashi. Type-based automated verification of authenticity in cryptographic protocols. In *Proceedings of ESOP 2009*, volume 5502, pages 222–236, 2009.

[Mil99] Robin Milner. *Communicating and Mobile Systems - The Pi Calculus*. Cambridge University Press, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts i and ii. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[PC07] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology*, volume 4695 of *Lecture Notes in Computer Science*, pages 184–199. Springer Berlin Heidelberg, 2007.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 1997.

[PW05] Frank Puhlmann and Mathias Weske. Using the π-calculus for formalizing workflow patterns. In *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin Heidelberg, 2005.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench - a tool for the π-calculus. In *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[WL93] Thomas Y.C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–193, 1993.

Chapter 2

Type-Based Verification of Authenticity

Gordon and Jeffrey developed a type system for verification of asymmetric and symmetric cryptographic protocols. We propose a modified version of their type system and develop a type inference algorithm for it, so that protocols can be verified automatically as they are, without any type annotations or explicit type casts. We have implemented a protocol verifier SpiCA2 based on the algorithm, and confirmed its effectiveness.

2.1 Introduction

Security protocols play a crucial role in today's Internet technologies including electronic commerce and voting. Formal verification of security protocols is thus an important, active research topic, and a variety of approaches to (semi-)automated verification have been proposed [Cre08, Bla02, GJ04]. Among others, type-based approaches [Aba99, GJ03, GJ04] have advantages that protocols can be verified in a modular manner, and that it is relatively easy to extend them to verify protocols at the source code level [BFG10]. They have however a disadvantage that users have to provide complex type annotations, which require expertise in both security protocols and type theories. Kikuchi and Kobayashi [KK09] developed a type inference algorithm but it works only for symmetric cryptographic protocols.

To overcome the limitation of the type-based approaches and enable fully automated protocol verification, we integrate and extend the two lines of work – Gordon and Jeffrey's work [GJ04] for verifying protocols using both symmetric and asymmetric cryptographic protocols, and Kikuchi and Kobayashi's work. The outcome is an algorithm for automated verification of authenticity in symmetric and asymmetric cryptographic protocols. The key technical novelty lies in the symmetric notion of *obligations* and *capabilities* attached to name types, which allows us to reason about causalities between actions of protocol participants in a general and uniform manner in the type system. It not only enables automated type inference, but also brings a more expressive power, enabling, e.g., verification of cryptographic protocols with several parties. We have developed a type inference algorithm for the new type system, and implemented a protocol verification tool SpiCA2 based on the algorithm. According to experiments, SpiCA2 is very fast; it could successfully verify a number of protocols in less than a second.

The rest of this chapter is structured as follows. Section 2.2 introduces a spi-calculus [AG99] extended with correspondence assertions as a protocol description language. Sections 2.3 and 2.4 present our type system and sketches a type inference algorithm. Section 2.5 reports implementation and experiments. Sections 2.6 and 2.7 discuss extensions and related work respectively. Section 2.8 compares the expressive power of our type system with that of Gordon and Jeffrey's and Section 2.9 gives the full proofs.

POSH:	**SOPH:**	**SOSH:**
`A->B: n`	`A->B:` $\{\lvert$`(msg,n)`$\rvert\}_{pk_B}$	`A->B:` $\{\lvert$`n`$\rvert\}_{pk_B}$
`B begins msg`	`B begins msg`	`B begins msg`
`B->A:` $\{\lvert$`(msg,n)`$\rvert\}_{sk_B}$	`B->A: n`	`B->A:` $\{\lvert$`msg,n`$\rvert\}_{pk_A}$
`A ends msg`	`A ends msg`	`A ends msg`

Figure 2.1: Informal Description of Three Protocols

2.2 Processes

This section defines the syntax and operational semantics of the spi-calculus [AG99] extended with correspondence assertions, which we call spi$_{CA}$. The calculus is essentially the same as that of Gordon and Jeffrey [GJ04], except (i) there are no type annotations or casts (as they can be automatically inferred by our type inference algorithm), and (ii) there are no primitives for witness and trust; supporting them is left for future work.

We assume that there is a countable set of *names*, ranged over by $m, n, k, x, y, z, \ldots$. By convention, we often use $k, m, n, \ldots$ for free names and $x, y, z, \ldots$ for bound names.

The set of messages, ranged over by M, is given by:

$$M ::= x \mid (M_1, M_2) \mid \{M_1\}_{M_2} \mid \{\!|M_1|\!\}_{M_2}$$

(M_1, M_2) is a pair consisting of M_1 and M_2. The message $\{M_1\}_{M_2}$ ($\{\!|M_1|\!\}_{M_2}$, resp.) represents the ciphertext obtained by encrypting M_1 with the symmetric (asymmetric, resp.) key M_2. For the asymmetric encryption, we do not distinguish between encryption and signing; $\{\!|M_1|\!\}_{M_2}$ denotes an encryption if M_2 is a public key, while it denotes signing if M_2 is a private key.

The set of processes, ranged over by P, is given by:

$$\begin{aligned} P ::= {} & \mathbf{0} \mid M_1!M_2 \mid M?x.P \mid (P_1 \mid P_2) \mid *P \mid (\nu x)P \mid (\nu_{sym} x)P \mid (\nu_{asym} x, y)P \\ & \mid \textbf{check } M_1 \textbf{ is } M_2.P \mid \textbf{split } M \textbf{ is } (x, y).P \mid \textbf{match } M_1 \textbf{ is } (M_2, y).P \\ & \mid \textbf{decrypt } M_1 \textbf{ is } \{x\}_{M_2}.P \mid \textbf{decrypt } M_1 \textbf{ is } \{\!|x|\!\}_{M_2{}^{-1}}.P \\ & \mid \textbf{begin } M.P \mid \textbf{end } M \end{aligned}$$

The names denoted by x, y are *bound* in P. We write $[M_1/x_1, \ldots, M_n/x_n]P$ for the process obtained by replacing every free occurrence of $x_1, \ldots, x_n$ in P with $M_1, \ldots, M_n$. We write $\mathbf{FN}(P)$ for the set of free (i.e. non-bounded) names in P.

Process $\mathbf{0}$ does nothing, $M_1!M_2$ sends M_2 over the channel M_1, and $M_1?x.P$ waits to receive a message on channel M_1, and then binds x to it and behaves like P. $P_1 \mid P_2$ executes P_1 and P_2 in parallel, and $*P$ executes infinitely many copies of P in parallel.

We have three kinds of name generation primitives: (νx) for ordinary names, $(\nu_{sym} x)$ for symmetric keys, and $(\nu_{asym} x_1, x_2,)$ for asymmetric keys. $(\nu_{asym} x_1, x_2, P)$ creates a fresh key pair (k_1, k_2) (where k_1 and k_2 are encryption and decryption keys respectively), and behaves like $[k_1/x_1, k_2/x_2]P$. The process **check** M_1 **is** $M_2.P$ behaves like P if M_1 and M_2 are the same name, and otherwise behaves like $\mathbf{0}$. The process **split** M **is** $(x, y).P$ behaves like $[M_1/x, M_2/y]P$ if M is a pair (M_1, M_2); otherwise it behaves like $\mathbf{0}$. **match** M_1 **is** $(M_2, y).P$ behaves like $[M_3/y]P$ if M_1 is a pair of the form (M_2, M_3); otherwise it behaves like $\mathbf{0}$. Process **decrypt** M_1 **is** $\{x\}_{M_2}.P$ (**decrypt** M_1 **is** $\{\!|x|\!\}_{M_2{}^{-1}}.P$, resp.) decrypts ciphertext M_1 with symmetric (asymmetric, resp.) key M_2, binds x to the result and behaves like P; if M_1 is not an encryption, or an encryption with a key not matching M_2, then it behaves like $\mathbf{0}$. The process **begin** $M.P$ raise an event **begin** M and behaves like P, while **end** M just raises an event **end** M; they are used to express expected authenticity properties.

Example 2.2.1. *We use the three protocols in Figure 2.1, taken from [GJ04], as running examples. The POSH and SOSH protocols aim to pass a new message* `msg` *from B to A, so*

$(\nu_{asym}\, sk_B, pk_B)(net!pk_B \mid$ (* create asymmetric keys for B and make pk_B public *)
$(\nu non)(net!non \mid$ (* A creates a nonce and sends it *)
$net?ctext.$**decrypt** $ctext$ **is** $\{\!|x|\!\}_{pk_B{}^{-1}}.$ (* receive a cypertext and decrypt it*)
split x **is** $(m, non').$**check** non **is** $non'.$ (* decompose pair x and check nonce *)
end $m) \mid$ (* believe that m came from B *)
$net?n.$ (* B receives a nonce *)
(νmsg)**begin** $msg.$ (* create a message and declare that it is going to be sent*)
$net!\{\!|(msg, n)|\!\}_{sk_B})$ (* encrypt and send (msg, n) *)

Figure 2.2: Public-Out-Secret-Home (POSH) Protocol in spi_{CA}

$(\nu_{asym}\, pk_B, sk_B)$ (* create asymmetric keys for B *)
$(net!pk_B$ (* make pk_B public *)
$\mid$ (* Behavior of A *)
$(\nu non)(\nu msg)$ (* create a nonce and a message *)
$(net!\{\!|(msg, non)|\!\}_{pk_B} \mid$ (* encrypt and send (msg, non) *)
$net?non'.$ (* receive a nonce *)
check non **is** $non'.$ (* check nonce *)
end $msg)$ (* end assertion *)
$\mid$ (* Behavior of B *)
$net?ctext.$ (* receive a cypertext *)
decrypt $ctext$ **is** $\{\!|x|\!\}_{sk_B{}^{-1}}.$ (* decrypt the cypertext *)
split x **is** $(m, non'').$ (* decompose pair x *)
begin $m.$ (* begin assertion *)
$net!non'')$ (* send the nonce *)

Figure 2.3: Secret-Out-Public-Home (SOPH) Protocol in spi_{CA}

that A can confirm that `msg` *indeed comes from B, while the SOPH protocol aims to pass* `msg` *from A to B, so that A can confirm that* `msg` *has been received by B. The second and fourth lines of each protocol expresses the required authenticity by using Woo and Lam's correspondence assertions [WL93]. "*`B begins msg`*" on the second line of POSH means "B is going to send* `msg`*", and "*`A ends msg`*" on the fourth line means "A believes that B has sent* `msg`*". The required authenticity is then expressed as a correspondence between begin- and end-events: whenever an end-event ("*`A ends msg`*" in this example) occurs, the corresponding begin-event ("*`B begins msg`*") must have occurred.*[1] *In the three protocols, the correspondence between begin- and end-events is guaranteed in different ways. In POSH, the correspondence is guaranteed by the signing of the second message with B's secret key, so that A can verify that B has created the pair (msg, n). In SOPH, it is guaranteed by encrypting the first message with B's public key, so that the nonce n, used as an acknowledgment, cannot be forged by an attacker. SOSH is similar to POSH, but keeps* `n` *secret by using A and B's public keys.*

Figure 2.2 gives a formal description of the POSH protocol, represented as a process in spi_{CA}. The first line is an initial set-up for the protocol. An asymmetric key pair for B is created and the decryption key pk_B is sent on a public channel net, on which an attacker can send and receive messages. The next four lines describe the behavior of A. On the second line, a nonce non is created and sent along net. On the third line, a ciphertext ctext is received and decrypted (or verified) with B's public key. On the fourth line, the pair is decomposed and it is checked that the second component coincides with the nonce sent before. On the fifth line,

[1] There are two types of correspondence assertions in the literature: non-injective (or one-to-many) and injective (or one-to-one) correspondence. Throughout the chapter we consider the latter.

an end-event is raised, meaning that A believes that msg came from B. The last three lines describe the behavior of B. On the sixth line, a nonce n is received from net. On the seventh line, a new message msg is created and a begin-event is raised, meaning that B is going to send msg. On the last line, the pair (msg, n) is encrypted (or signed) with B's secret key and sent on net.

Figure 2.3 gives a formal description of the SOPH protocol in spi_{CA}. □

Following Gordon and Jeffrey, we call a process *safe* if it satisfies correspondence assertions (i.e. for each end-event, a corresponding begin-event has occurred before), and *robustly safe* if a process is safe in the presence of arbitrary attackers (representable in spi_{CA}). Proving robust safety automatically is the goal of protocol verification in the present chapter. To formalize the robust safety, we use the operational semantics shown in Figure 2.4. A runtime state is a quadruple $\langle \Psi, E, N, \mathcal{K} \rangle$, where Ψ is a multiset of processes, and E is the set of messages on which begin-events have occurred but the matching end-events have not. N is the set of names (including keys) created so far, and $\mathcal{K}$ is the set of key pairs. The special runtime state **Error** denotes that correspondence assertions have been violated. Note that a reduction gets stuck when a process does not match a rule. For example, **split** M **is** $(x, y).P$ is reducible only if M is of the form (M_1, M_2). Using the operational semantics, robust safety is defined as follows:

Definition 2.2.1 (safety, robust safety)**.** *A process P is* safe *if $\langle \{P\}, \emptyset, \mathbf{FN}(P), \emptyset \rangle \not\longrightarrow^* \mathbf{Error}$. A process P is* robustly safe *if $P|O$ is safe for every spi_{CA} process O that contains no begin/end/check operations.*[2]

$$\langle \Psi \uplus \{n?y.P, n!M\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[M/y]P\}, E, N, \mathcal{K} \rangle \quad \text{(R-Com)}$$

$$\langle \Psi \uplus \{P \mid Q\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{P, Q\}, E, N, \mathcal{K} \rangle \quad \text{(R-Par)}$$

$$\langle \Psi \uplus \{*P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{*P, P\}, E, N, \mathcal{K} \rangle \quad \text{(R-Rep)}$$

$$\langle \Psi \uplus \{(\nu x)P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[n/x]P\}, E, N \cup \{n\}, \mathcal{K} \rangle \ (n \notin N) \quad \text{(R-New)}$$

$$\langle \Psi \uplus \{(\nu_{sym} x)P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[k/x]P\}, E, N \cup \{k\}, \mathcal{K} \rangle \ (k \notin N) \quad \text{(R-NewSk)}$$

$$\langle \Psi \uplus \{(\nu_{asym} x, y)P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[k_1/x, k_2/y]P\}, E, N \cup \{k_1, k_2\}, \mathcal{K} \cup \{(k_1, k_2)\} \rangle \ (k_1, k_2 \notin N) \quad \text{(R-NewAk)}$$

$$\langle \Psi \uplus \{\mathbf{check}\ n\ \mathbf{is}\ n.P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{P\}, E, N, \mathcal{K} \rangle \quad \text{(R-Chk)}$$

$$\langle \Psi \uplus \{\mathbf{split}\ (M, N)\ \mathbf{is}\ (x, y).P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[M/x, N/y]P\}, E, N, \mathcal{K} \rangle \quad \text{(R-Splt)}$$

$$\langle \Psi \uplus \{\mathbf{match}\ (M, N)\ \mathbf{is}\ (M, z).P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[N/z]P\}, E, N, \mathcal{K} \rangle \quad \text{(R-Mtch)}$$

$$\langle \Psi \uplus \{\mathbf{decrypt}\ \{M\}_k\ \mathbf{is}\ \{x\}_k.P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[M/x]P\}, E, N, \mathcal{K} \rangle \quad \text{(R-DecS)}$$

$$\langle \Psi \uplus \{\mathbf{decrypt}\ \{\!|M|\!\}_{k_1}\ \mathbf{is}\ \{\!|x|\!\}_{k_2{}^{-1}}.P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{[M/x]P\}, E, N, \mathcal{K} \rangle \ (\text{if } (k_1, k_2) \in \mathcal{K}) \quad \text{(R-DecA)}$$

$$\langle \Psi \uplus \{\mathbf{begin}\, M.P\}, E, N, \mathcal{K} \rangle \longrightarrow \langle \Psi \uplus \{P\}, E \uplus \{M\}, N, \mathcal{K} \rangle \quad \text{(R-Bgn)}$$

$$\langle \Psi \uplus \{\mathbf{end}\, M\}, E \uplus \{M\}, N, \mathcal{K} \rangle \longrightarrow \langle \Psi, E, N, \mathcal{K} \rangle \quad \text{(R-End)}$$

$$\langle \Psi \uplus \{\mathbf{end}\, M\}, E, N, \mathcal{K} \rangle \longrightarrow \mathbf{Error} \quad (\text{if } M \notin E) \quad \text{(R-Err)}$$

Figure 2.4: Operational Semantics

2.3 Type System

This section presents a type system such that well-typed processes are robustly safe. This allows us to reduce protocol verification to type inference.

[2]Having no check operations is not a limitation, as an attacker process can check the equality of n_1 and n_2 by **match** (n_1, n_1) **is** $(n_2, x).P$.

2.3.1 Basic Ideas

Following the previous work [GJ03, GJ04, KK09], we use the notion of *capabilities* (called effects in [GJ03, GJ04]) in order to statically guarantee that end-events can be raised only after the corresponding begin-events. A capability φ is a multiset of *atomic capabilities* of the form $\mathbf{end}(M)$, which expresses a permission to raise "end M" event. The robust safety of processes is guaranteed by enforcing the following conditions on capabilities: (i) to raise an "end M" event, a process must possess and consume an atomic $\mathbf{end}(M)$ capability; and (ii) an atomic $\mathbf{end}(M)$ capability is generated only by raising a "begin M" event. Those conditions can be statically enforced by using a type judgment of the form: $\Gamma; \varphi \vdash P$, which means that P can be safely executed under the type environment Γ and the capabilities described by φ. For example, $x : T; \{\mathbf{end}(x)\} \vdash \mathbf{end}\, x$ is a valid judgment, but $x : T; \emptyset \vdash \mathbf{end}\, x$ is not. The two conditions above can be locally enforced by the following typing rules for begin and end events:

$$\frac{\Gamma; \varphi + \{\mathbf{end}(M)\} \vdash P}{\Gamma; \varphi \vdash \mathbf{begin}\, M.P} \qquad\qquad \frac{}{\Gamma; \varphi + \{\mathbf{end}(M)\} \vdash \mathbf{end}\, M}$$

The left rule ensures that the new capability $\mathbf{end}(M)$ is available after the begin-event, and the right rule for end ensures that the capability $\mathbf{end}(M)$ must be present.

The main difficulty lies in how to pass capabilities between processes. For example, recall the POSH protocol in Figure 2.2, where begin- and end-events are raised by different protocol participants. The safety of this protocol can be understood as follows: B obtains the capability $\mathbf{end}(msg)$ by raising the begin event, and then passes the capability to A by attaching it to the nonce n. A then extracts the capability and safely executes the end event. As n is signed with B's private key, there is no way for an attacker to forge the capability. For another example, consider the SOPH protocol in the middle of Figure 2.1. In this case, the nonce n is sent in clear text, so that B cannot pass the capability to A through the second message. Instead, the safety of the SOPH protocol is understood as follows: A attaches to n (in the first message) an *obligation* to raise the begin-event. B then discharges the obligation by raising the begin-event, and notifies of it by sending back n. Here, note that an attacker cannot forge n, as it is encrypted by B's public key in the first message.

To capture the above reasoning by using types, we introduce types of the form $\mathbf{N}(\varphi_1, \varphi_2)$, which describes names carrying an obligation φ_1 and a capability φ_2. In the examples above, n is given the type $\mathbf{N}(\emptyset, \{\mathbf{end}(msg)\})$ in the second message of the POSH protocol, and the type $\mathbf{N}(\{\mathbf{end}(msg)\}, \emptyset)$ in the first message of the SOPH protocol.

The above types $\mathbf{N}(\emptyset, \{\mathbf{end}(msg)\})$ and $\mathbf{N}(\{\mathbf{end}(msg)\}, \emptyset)$ respectively correspond to *response* and *challenge types* in Gordon and Jeffrey's type system [GJ04]. Thanks to the uniform treatment of name types, type inference for our type system reduces to a problem of solving constraints on capabilities and obligations, which can further be reduced to linear programming problems by using the technique of [KK09]. The uniform treatment also allows us to express a wider range of protocols (such as cryptographic protocols with several parties). Note that neither obligations nor asymmetric cryptography are supported by the previous type system for automated verification [KK09]; handling them requires non-trivial extensions of the type system and the inference algorithm.

2.3.2 Types

The syntax of types, ranged over by τ, is given in Figure 2.5, where r_i ranges over non-negative rational numbers. The type $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ is assigned to names carrying obligations φ_1 and capabilities φ_2. Here, obligations and capabilities are mappings from atomic capabilities to rational numbers. For example, $\mathbf{N}_\ell(\{\mathbf{end}(a) \mapsto 1.0\}, \{\mathbf{end}(b) \mapsto 2.0\})$ describes a name that carries the obligation to raise $\mathbf{begin}\, a$ once, and the capability to raise $\mathbf{end}\, b$ twice. Fractional

$$
\begin{array}{rr}
\tau ::= \mathbf{N}_\ell(\varphi_1, \varphi_2) \mid \mathbf{SKey}(\tau) \mid \mathbf{DKey}(\tau) \mid \mathbf{EKey}(\tau) \mid \tau_1 \times \tau_2 & \text{types} \\
\varphi ::= \{A_1 \mapsto r_1, \ldots, A_m \mapsto r_m\} & \text{capabilities} \\
A ::= \mathbf{end}(M) \mid \mathbf{chk}_\ell(M, \varphi) & \text{atomic capabilities} \\
\iota ::= x \mid 0 \mid 1 \mid 2 \mid \cdots & \text{extended names} \\
\ell ::= \mathbf{Pub} \mid \mathbf{Pr} & \text{name qualifiers}
\end{array}
$$

Figure 2.5: Types and Capabilities

values are possible: $\mathbf{N}_\ell(\emptyset, \{\mathbf{end}(b) \mapsto 0.5\})$ means that the name carries a half of the capability to raise $\mathbf{end}\, b$, so that if combined with another half of the capability, it is allowed to raise $\mathbf{end}\, b$. The introduction of fractions slightly increases the expressive power of the type system, but the main motivation for it is rather to enable efficient type inference as in [KK09]. When the ranges of obligations and capabilities are integers, we often use multiset notations; for example, we write $\{\mathbf{end}(a), \mathbf{end}(a), \mathbf{end}(b)\}$ for $\{\mathbf{end}(a) \mapsto 2, \mathbf{end}(b) \mapsto 1\}$. The atomic capability $\mathbf{chk}_\ell(M, \varphi)$ expresses the capability to check equality on M by **check** M **is** $M'.P$: since nonce checking releases capabilities this atomic effect is used to ensure that each nonce can only be checked once. The component φ expresses the capability that can be extracted by the check operation (see the typing rule for check operations given later).

Qualifier ℓ attached to name types are essentially the same as the **Public**/**Private** qualifiers in Gordon and Jeffrey's type system and express whether a name can be made public or not. We often write **Un** for $\mathbf{N}_{\mathbf{Pub}}(\emptyset, \emptyset)$.

The type $\mathbf{SKey}(\tau)$ describes symmetric keys used for decrypting and encrypting values of type τ. The type $\mathbf{EKey}(\tau)$ ($\mathbf{DKey}(\tau)$, resp.) describes asymmetric keys used for encrypting (decrypting, resp.) values of type τ. The type $\tau_1 \times \tau_2$ describes pairs of values of types τ_1 and τ_2. As in [KK09], we express the dependency of types on names by using indices. For example, the type $\mathbf{Un} \times \mathbf{N}_\ell(\emptyset, \{\mathbf{end}(0)\})$ denotes a pair (M_1, M_2) where M_1 has type **Un** and M_2 has type $\mathbf{N}_\ell(\emptyset, \{\mathbf{end}(M_1)\})$. The type $\mathbf{Un} \times (\mathbf{Un} \times \mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(0, 1) \mapsto r\})$ describes triples of the form $(M_1, (M_2, M_3))$, where M_1 and M_2 have type **Un**, and M_3 has type $\mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(M_2, M_1) \mapsto r\})$. In general, an index i is a natural number referring to the i-th closest first component of pairs. In the syntax of atomic capabilities $\mathbf{end}(M)$, M is an extended message that may contain indices. We use the same metavariable M for the sake of simplicity.

Predicates on types

Following Gordon and Jeffrey, we introduce two predicates **Pub** and **Taint** on types, inductively defined by the rules in Figure 2.6. $\mathbf{Pub}(\tau)$ means that a value of type τ can safely be made public by e.g. sending it through a public channel. $\mathbf{Taint}(\tau)$ means that a value of type τ may have come from an untrusted principal and hence cannot be trusted. It may for instance have been received through a public channel or have been extracted from a ciphertext encrypted with a public key.

The first rule says that for $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ to be public, the obligation φ_1 must be empty, as there is no guarantee that an attacker fulfills the obligation. Contrary, for $\mathbf{N}_\ell(\varphi_1, \varphi_2)$ to be tainted, the capability φ_2 must be empty if $\ell = \mathbf{Pub}$, as the name may come from an attacker and the capability cannot be trusted.[3]

Pub and **Taint** are a sort of dual, flipped by the type constructor **EKey**. In terms of subtyping, $\mathbf{Pub}(\tau)$ and $\mathbf{Taint}(\tau)$ may be understood as $\tau \leq \mathbf{Un}$ and $\mathbf{Un} \leq \tau$ respectively,

[3] These conditions are more liberal than the corresponding conditions in Gordon and Jeffrey's type system. In their type system, for Public Challenge φ_1 (which corresponds to $\mathbf{N}_{\mathbf{Pub}}(\varphi_1, \emptyset)$ in our type system) to be tainted, φ_1 must also be empty.

$$\frac{\ell = \mathbf{Pub} \qquad \varphi_1 = \emptyset}{\mathbf{Pub}(\mathbf{N}_\ell(\varphi_1, \varphi_2))} \qquad \frac{\ell = \mathbf{Pub} \Rightarrow \varphi_2 = \emptyset}{\mathbf{Taint}(\mathbf{N}_\ell(\varphi_1, \varphi_2))} \qquad \frac{\mathbf{Pub}(\tau_1) \qquad \mathbf{Pub}(\tau_2)}{\mathbf{Pub}(\tau_1 \times \tau_2)}$$

$$\frac{\mathbf{Taint}(\tau_1) \qquad \mathbf{Taint}(\tau_2)}{\mathbf{Taint}(\tau_1 \times \tau_2)} \qquad \frac{\mathbf{Pub}(\tau) \qquad \mathbf{Taint}(\tau)}{\mathbf{Pub}(\mathbf{SKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau) \qquad \mathbf{Taint}(\tau)}{\mathbf{Taint}(\mathbf{SKey}(\tau))}$$

$$\frac{\mathbf{Taint}(\tau)}{\mathbf{Pub}(\mathbf{EKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau)}{\mathbf{Taint}(\mathbf{EKey}(\tau))} \qquad \frac{\mathbf{Pub}(\tau)}{\mathbf{Pub}(\mathbf{DKey}(\tau))} \qquad \frac{\mathbf{Taint}(\tau)}{\mathbf{Taint}(\mathbf{DKey}(\tau))}$$

Figure 2.6: Predicates **Pub** and **Taint**

where **Un** is the type of untrusted, non-secret data. Note that **DKey** is co-variant, **EKey** is contra-variant, and **SKey** is invariant; this is analogous to Pierce and Sangiorgi's IO types with subtyping [PS96].

Operations and relations on capabilities and types

We write $dom(\varphi)$ for the set $\{A \mid \varphi(A) > 0\}$. We identify capabilities up to the following equality $\approx$:

$$\varphi_1 \approx \varphi_2 \iff (dom(\varphi_1) = dom(\varphi_2) \wedge \forall A \in dom(\varphi_1).\varphi_1(A) = \varphi_2(A)).$$

We write $\varphi \leq \varphi'$ if $\varphi(A) \leq \varphi'(A)$ holds for every $A \in dom(\varphi)$ and we define the summation of two capabilities by: $(\varphi_1 + \varphi_2)(A) = \varphi_1(A) + \varphi_2(A)$. This is a natural extension of the multiset union. We write $\varphi_1 - \varphi_2$ for the least φ such that $\varphi_1 \leq \varphi + \varphi_2$.

As we use indices to express dependent types, messages may be substituted in types. Let i be an index and M a message. The substitution $[M/i]\tau$ is defined inductively in the straightforward manner, except for pair types where

$$[M/i](\tau_1 \times \tau_2) = ([M/i]\tau_1) \times ([M/(i+1)]\tau)$$

such that the index is shifted for the second component.

2.3.3 Typing

We introduce two forms of type judgments: $\Gamma; \varphi \vdash M : \tau$ for messages, and $\Gamma; \varphi \vdash P$ for processes, where Γ, called a type environment, is a sequence of type bindings of the form $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. Judgment $\Gamma; \varphi \vdash M : \tau$ means that M evaluates to a value of type τ under the assumption that each name has the type described by Γ and that capability φ is available. $\Gamma; \varphi \vdash P$ means that P can be safely executed (i.e. without violation of correspondence assertions) if each free name has the type described by Γ and the capability φ is available. For example, $x : \mathbf{Un}; \{\mathbf{end}(x)\} \vdash \mathbf{end}\, x$ is valid but $x : \mathbf{Un}; \emptyset \vdash \mathbf{end}\, x$ is not.

We consider only the judgements that are *well-formed* in the sense that (i) φ refers to only the names bound in Γ, and (ii) Γ must be well-formed, i.e., if Γ is of the form $\Gamma_1, x : \tau, \Gamma_2$ then τ only refers to the names bound in Γ_1 and x is not bound in neither Γ_1 nor Γ_2. Formally, the conditions for type judgments and type environments are given in Figure 2.7. Here, $\uparrow N$ denotes the set of extended names obtained from N by replacing each number i in N with $i+1$. For example, $\uparrow\{x, y, 0\} = \{x, y, 1\}$. We freely permute bindings in type environments as long as they are well-formed; for example, we do not distinguish between $x : \mathbf{Un}, y : \mathbf{Un}$ and $y : \mathbf{Un}, x : \mathbf{Un}$.

$$\frac{\vdash_{\mathtt{wf}} \Gamma \qquad \mathbf{FN}(\varphi) \subseteq dom(\Gamma) \qquad dom(\Gamma) \vdash_{\mathtt{wf}} \tau}{\vdash_{\mathtt{wf}} \Gamma; \varphi \vdash M : \tau} \qquad \frac{\vdash_{\mathtt{wf}} \Gamma \qquad \mathbf{FN}(\varphi) \subseteq dom(\Gamma)}{\vdash_{\mathtt{wf}} \Gamma; \varphi \vdash P}$$

$$\frac{}{\vdash_{\mathtt{wf}} \emptyset} \qquad \frac{\vdash_{\mathtt{wf}} \Gamma \qquad dom(\Gamma) \vdash_{\mathtt{wf}} \tau \qquad x \notin dom(\Gamma)}{\vdash_{\mathtt{wf}} \Gamma, x : \tau} \qquad \frac{N \vdash_{\mathtt{wf}} \tau_1 \qquad \{0\} \cup \uparrow N \vdash_{\mathtt{wf}} \tau_2}{N \vdash_{\mathtt{wf}} \tau_1 \times \tau_2}$$

$$\frac{\mathbf{FN}(\varphi_1) \cup \mathbf{FN}(\varphi_2) \subseteq N}{N \vdash_{\mathtt{wf}} \mathbf{N}_\ell(\varphi_1, \varphi_2)} \qquad \frac{N \vdash_{\mathtt{wf}} \tau}{N \vdash_{\mathtt{wf}} \mathbf{SKey}(\tau)} \qquad \frac{N \vdash_{\mathtt{wf}} \tau}{N \vdash_{\mathtt{wf}} \mathbf{DKey}(\tau)} \qquad \frac{N \vdash_{\mathtt{wf}} \tau}{N \vdash_{\mathtt{wf}} \mathbf{EKey}(\tau)}$$

Figure 2.7: Well-formedness Conditions for Type Judgments

Typing

The typing rules are shown in Figure 2.8. The rule T-Cast says that the current capability can be used for discharging obligations and increasing capabilities of the name. T-Cast plays a role similar to the typing rule for cast processes in Gordon and Jeffrey's type system, but our cast is implicit and changes only the capabilities and obligations, not the shape of types. This difference is important for automated type inference. The other rules for messages are standard; T-Pair is the standard rule for dependent sum types (except for the use of indices).

In the rules for processes, the capabilities shown by _ can be any capabilities. The rules are also similar to those of Gordon and Jeffrey, except for the rules T-Out, T-In, T-NewN, and T-Chk. In rule T-Out, we require that the type of message M_2 is public as it can be received by any process, including the attacker. Similarly, in rule T-In we require that the type of the received value x is tainted, as it may come from any process. This is different from Gordon and Jeffrey's type system where the type of messages sent to or received from public channels must be **Un**, and a subsumption rule allows any value of a public type to be typed as **Un** and a value of type **Un** to be typed as any tainted type. In effect, our type system can be considered a restriction of Gordon and Jeffrey's such that the subsumption rule is only allowed for messages sent or received via public channels. This point is important for automated type inference.

In rule T-NewN, the obligation φ_1 is attached to the fresh name x and recorded in the atomic check capability. Capabilities corresponding to φ_1 can then later be extracted by a check operation if the obligation has been fulfilled. In rule T-Chk, $\mathbf{chk}_\ell(M_1, \varphi_4)$ in the conclusion means that the capability to check M_1 must be present. If the check succeeds, the capability φ_5 attached to M_2 can be extracted and used in P. In addition, the obligations attached to M_2 must be empty, i.e. all obligations initially attached to the name must have been fulfilled, and hence the capability φ_4 can be extracted and used in P. The above mechanism for extracting capabilities through obligations is different from Gordon and Jeffrey's type system in a subtle but important way, and provides more expressive power (see Section 2.8). The remaining rules should be self-explanatory.

Example 2.3.1. *Recall the POSH protocol in Figure 2.2. Let* τ *be* $\mathbf{Un} \times \mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(0)\})$. *Then the process describing the behavior of B (net?n.* $\cdots$ *in the last five lines) is typed as the upper part of Figure 2.9. Here,* $\Gamma = net{:}\mathbf{Un}, sk_B{:}\mathbf{EKey}(\tau), n{:}\mathbf{Un}, msg{:}\mathbf{Un}$. *Similarly, the part* **decrypt** *ctext* **is** $\{\!|x|\!\}_{pk_B{}^{-1}}.\cdots$ *of process A is typed as the lower part of Figure 2.9. Here,* $\Gamma_2 = net{:}\mathbf{Un}, pk_B{:}\mathbf{DKey}(\tau), non{:}\mathbf{Un}, ctext{:}\mathbf{Un}$ *and* $\Gamma_3 = \Gamma_2, x{:}\tau, m{:}\mathbf{Un}, non'{:}\mathbf{N}_{\mathbf{Pub}}(\emptyset, \{\mathbf{end}(m)\})$. *Let* P_1 *be the entire process of the POSH protocol. It is typed by* $net : \mathbf{Un}; \emptyset \vdash P_1$.

The SOPH and SOSH protocols in Figure 2.1 are typed in a similar manner. We show here

$$\frac{}{\Gamma, x:\tau; \varphi \vdash x:\tau}\ (\text{T-Var}) \qquad \frac{\Gamma;\varphi_1 \vdash M_1:\tau_1 \qquad \Gamma;\varphi_2 \vdash M_2:[M_1/0]\tau_2}{\Gamma;\varphi_1+\varphi_2 \vdash (M_1,M_2):\tau_1\times\tau_2}\ (\text{T-Pair})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\tau_1 \quad \Gamma;\varphi_2 \vdash M_2:\mathbf{SKey}(\tau_1)}{\Gamma;\varphi_1+\varphi_2 \vdash \{M_1\}_{M_2}:\mathbf{N}_\ell(\emptyset,\emptyset)}\ (\text{T-SEnc}) \qquad \frac{\Gamma;\varphi_1 \vdash M_1:\tau \quad \Gamma;\varphi_2 \vdash M_2:\mathbf{EKey}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash \{\!|M_1|\!\}_{M_2}:\mathbf{N}_\ell(\emptyset,\emptyset)}\ (\text{T-AEnc})$$

$$\frac{}{\Gamma;\emptyset \vdash \mathbf{0}}\ (\text{T-Zero}) \qquad \frac{\Gamma;\varphi_1 \vdash M:\mathbf{N}_\ell(\varphi_2,\varphi_3)}{\Gamma;\varphi_1+\varphi_2'+\varphi_3' \vdash M:\mathbf{N}_\ell(\varphi_2-\varphi_2',\varphi_3+\varphi_3')}\ (\text{T-Cast})$$

$$\frac{\Gamma;\varphi_1 \vdash P_1 \qquad \Gamma;\varphi_2 \vdash P_2}{\Gamma;\varphi_1+\varphi_2 \vdash P_1 \mid P_2}\ (\text{T-Par}) \qquad \frac{\Gamma;\varphi' \vdash P \qquad \varphi' \le \varphi}{\Gamma;\varphi \vdash P}\ (\text{T-CSub})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\mathbf{N}_\ell(\emptyset,\emptyset) \quad \Gamma;\varphi_2 \vdash M_2:\tau \qquad \mathbf{Pub}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash M_1!M_2}\ (\text{T-Out}) \qquad \frac{\Gamma;\varphi_1 \vdash M:\mathbf{N}_\ell(\emptyset,\emptyset) \quad \Gamma, x:\tau;\varphi_2 \vdash P \quad \mathbf{Taint}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash M?x.P}\ (\text{T-In})$$

$$\frac{\Gamma, x:\mathbf{N}_\ell(\varphi_1,\emptyset), \varphi+\{\mathbf{chk}_\ell(x,\varphi_1)\} \vdash P}{\Gamma;\varphi \vdash (\nu x)P}\ (\text{T-NewN}) \qquad \frac{\Gamma;\emptyset \vdash P}{\Gamma;\emptyset \vdash *P}\ (\text{T-Rep})$$

$$\frac{\Gamma, x:\mathbf{SKey}(\tau);\varphi \vdash P}{\Gamma;\varphi \vdash (\nu_{sym} x)P}\ (\text{T-NewSk}) \qquad \frac{\Gamma, k_1:\mathbf{EKey}(\tau), k_2:\mathbf{DKey}(\tau);\varphi \vdash P}{\Gamma;\varphi \vdash (\nu_{asym} k_1,k_2)P}\ (\text{T-NewAk})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2:\mathbf{SKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P}\ (\text{T-SDec})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2:\mathbf{DKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2{}^{-1}}.P}\ (\text{T-ADec})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash M_2:\mathbf{N}_\ell(\emptyset,\varphi_5) \qquad \Gamma;\varphi_3+\varphi_4+\varphi_5 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3+\{\mathbf{chk}_\ell(M_1,\varphi_4)\} \vdash \mathbf{check}\ M_1\ \mathbf{is}\ M_2.P}\ (\text{T-Chk})$$

$$\frac{\Gamma;\varphi_1 \vdash M:\tau_1\times\tau_2 \qquad \Gamma, y:\tau_1, z:[y/0]\tau_2;\varphi_2 \vdash P}{\Gamma;\varphi_1+\varphi_2 \vdash \mathbf{split}\ M\ \mathbf{is}\ (y,z).P}\ (\text{T-Split})$$

$$\frac{\Gamma;\varphi_1 \vdash M_1:\tau_1\times\tau_2 \qquad \Gamma;\varphi_2 \vdash M_2:\tau_1 \qquad \Gamma, z:[M_2/0]\tau_2;\varphi_3 \vdash P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash \mathbf{match}\ M_1\ \mathbf{is}\ (M_2,z).P}\ (\text{T-Match})$$

$$\frac{\Gamma;\varphi+\{\mathbf{end}(M)\} \vdash P}{\Gamma;\varphi \vdash \mathbf{begin}\, M.P}\ (\text{T-Begin}) \qquad \frac{}{\Gamma;\varphi+\{\mathbf{end}(M)\} \vdash \mathbf{end}\, M}\ (\text{T-End})$$

Figure 2.8: Typing Rules

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma;\emptyset \vdash msg : \mathbf{Un} \quad \dfrac{\Gamma;\emptyset \vdash n : \mathbf{N_{Pub}}(\emptyset,\emptyset)}{\Gamma;\{\mathbf{end}(msg)\} \vdash n : \mathbf{N_{Pub}}(\emptyset,\{\mathbf{end}(msg)\})}}{\Gamma;\{\mathbf{end}(msg)\} \vdash (msg,n) : \tau}}{\cdots}}{\Gamma;\{\mathbf{end}(msg), \mathbf{chk_{Pub}}(msg,\emptyset)\} \vdash net!\{\!|(msg,n)|\!\}_{sk_B}}}{\Gamma;\{\mathbf{chk_{Pub}}(msg,\emptyset)\} \vdash \mathbf{begin}\ msg.\cdots}}{net : \mathbf{Un}, sk_B : \mathbf{EKey}(\tau), n : \mathbf{Un}; \emptyset \vdash (\nu msg)\cdots}}{net : \mathbf{Un}, sk_B : \mathbf{EKey}(\tau); \emptyset \vdash net?n.\cdots}$$

$$\dfrac{\dfrac{\dfrac{\Gamma_3;\{\mathbf{end}(m)\} \vdash \mathbf{end}\ m}{\Gamma_3;\{\mathbf{chk_{Pub}}(non,\emptyset)\} \vdash \mathbf{check}\ non\ \mathbf{is}\ non'.\cdots}}{\Gamma_2, x:\tau;\{\mathbf{chk_{Pub}}(non,\emptyset)\} \vdash \mathbf{split}\ x\ \mathbf{is}\ (m,non).\cdots}}{\Gamma_2;\{\mathbf{chk_{Pub}}(non,\emptyset)\} \vdash \mathbf{decrypt}\ ctext\ \mathbf{is}\ \{\!|x|\!\}_{pk_B{}^{-1}}.\cdots}$$

Figure 2.9: Partial Typing of the POSH Protocol

only key types:

SOPH:
$pk_B : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N_{Pub}}(\{\mathbf{end}(0)\},\emptyset)), \quad sk_B : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N_{Pub}}(\{\mathbf{end}(0)\},\emptyset))$

SOSH:
$pk_A : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N_{Pr}}(\emptyset,\{\mathbf{end}(0)\})), \quad sk_A : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N_{Pr}}(\emptyset,\{\mathbf{end}(0)\}))$
$pk_B : \mathbf{EKey}(\mathbf{Un} \times \mathbf{N_{Pr}}(\emptyset,\emptyset)), \quad sk_B : \mathbf{DKey}(\mathbf{Un} \times \mathbf{N_{Pr}}(\emptyset,\emptyset))$

Note that for POSH and SOPH the name qualifier must be **Pub**, *and only for the SOSH protocol may it be* **Pr**. □

2.3.4 Soundness of the Type System

We first prepare the following lemma, which implies that, in the definition of robust safety, it is sufficient to consider only well-typed opponent processes (see Section 2.9 for proofs):

Lemma 2.3.1. *If O is a process that contains no begin/end/check, then there exists O' that satisfies the following conditions:*

1. $x_1 : \mathbf{Un}, \ldots, x_m : \mathbf{Un}; \emptyset \vdash O'$, *where* $\{x_1, \ldots, x_k\} = \mathbf{FN}(O)$.
2. *For any process P, if $P \mid O'$ is safe then so is $P \mid O$.*

Hence, by the lemma above, it suffices to show the following lemma:

Lemma 2.3.2. *If $\emptyset;\emptyset \vdash P$ then P is safe.*

Soundness of the type system may then be stated as follows:

Theorem 2.3.3 (Soundness)**.** *If $x_1 : \mathbf{Un}, \ldots, x_m : \mathbf{Un}; \emptyset \vdash P$ then P is robustly safe.*

Proof. Suppose $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$. Let O be a process that does not contain begin/end/check. We need to show that $P \mid O$ is safe. By Lemma 2.3.1 there exists a process O' such that (i) $y_1 : \mathbf{Un}, \ldots, y_k : \mathbf{Un}; \emptyset \vdash O'$ and (ii) if $P \mid O'$ is safe then so is $P \mid O$. Let $\{z_1 : \mathbf{Un}, \ldots, z_m : \mathbf{Un}\} = \{x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}\} \cup \{y_1 : \mathbf{Un}, \ldots, y_k : \mathbf{Un}\}$. By weakening and the typing rules we have $\emptyset;\emptyset \vdash (\nu z_1)\cdots(\nu z_m)(P \mid O')$, and by Lemma 2.3.2 $(\nu z_1)\cdots(\nu z_m)(P \mid O')$ is safe. By definition of safety $P \mid O'$ is also safe, and by condition (ii) above so is $P \mid O$. □

2.4 Type Inference

We now briefly discuss type inference. For this we impose a minor restriction to the type system, namely that in rule T-Pair, if M_1 is not a name then the indice 0 cannot occur in τ_2. Similarly, in rule T-Match we require that index 0 does not occur unless M_2 is a name. These restrictions prevent the size of types and capabilities from blowing up. Given as input a process P with free names $x_1, \ldots, x_n$, the algorithm to decide $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ proceeds as follows:

1. Determine the *shape of the type* (or simple type) of each term via a standard unification algorithm, and construct a template of a type derivation tree by introducing qualifier and capability variables.
2. Generate a set C of constraints on qualifier and capability variables based on the typing rules such that C is satisfiable if and only if $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$.
3. Solve the qualifier constraints.
4. Transform the capability constraints to linear inequalities over the rational numbers.
5. Use linear programming to determine if the linear inequalities are satisfiable.

In step 1, we can assume that there are no consecutive applications of T-Cast and T-CSub. Thus, the template of a type derivation tree can be uniquely determined: for each process and message constructor there is an application of the rule matching the constructor followed by at most one application of T-Cast or T-CSub.

At step 3 we have a set of constraints C of the form:

$$\{\ell_i = \ell'_i \mid i \in I\} \cup \{(\ell''_j = \mathbf{Pub}) \Rightarrow (\varphi_j = \emptyset) \mid j \in J\} \cup C_1$$

where I and J are finite sets, $\ell_i, \ell'_i, \ell''_j$ are qualifier variables or constants, and C_1 is a set of effect constraints (like $\varphi_1 \leq \varphi_2$). Here, constraints on qualifiers come from equality constraints on types and conditions $\mathbf{Pub}(\tau)$ and $\mathbf{Taint}(\tau)$. In particular, $(\ell''_j = \mathbf{Pub}) \Rightarrow (\varphi_j = \emptyset)$ comes from the rule for $\mathbf{Taint}(\mathbf{N}_{\ell''_j}(\varphi, \varphi_j))$. By obtaining the most general unifier θ of the first set of constraints $\{\ell_i = \ell'_i \mid i \in I\}$ we obtain the constraint set $C' \equiv \{(\theta\ell''_j = \mathbf{Pub}) \Rightarrow (\theta\varphi_j = \emptyset) \mid j \in J\} \cup \theta C_1$. Let $\gamma_1, \ldots, \gamma_k$ be the remaining qualifier variables, and let $\theta' = [\mathbf{Pr}/\gamma_1, \ldots, \mathbf{Pr}/\gamma_k]$. Then C is satisfiable if and only if $\theta' C'$ is satisfiable. Thus, we obtain the set $\theta' C'$ of effect constraints that is satisfiable if and only if $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}; \emptyset \vdash P$ holds.

Except for step 3, the above algorithm is almost the same as previous work and we refer the interested reader to [KK07, KK09]. By a similar argument to that given in [KK09] we can show that under the assumptions that the size of each begin/end assertion occurring in the protocol is bounded by a constant and that the size of simple types is polynomial in the size of the protocol, the type inference algorithm runs in polynomial time.

Example 2.4.1. *Recall the POSH protocol in Figure 2.2. By the simple type inference in step 1 we get the following types for names:*

$$non, non' : \mathbf{N}, pk_B : \mathbf{DKey}(\mathbf{N} \times \mathbf{N}), \ldots$$

By preparing qualifier and capability variables we get the following elaborated types and constraints on those variables:

$$\begin{array}{l} non : \mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c}), non' : \mathbf{N}_{\gamma'_1}(\xi'_{0,o}, \xi'_{0,c}), \ldots \\ \mathbf{Pub}(\mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c})) \quad \gamma_1 = \gamma'_1 \quad \xi_6 \leq \xi_3 + \xi_4 + \xi_5 \\ \xi_2 \geq \xi'_{0,o} + (\xi_5 - \xi'_{0,c}) \quad \xi_7 \geq \xi_1 + \xi_2 + \xi_3 + \{\mathbf{chk}_{\gamma_1}(non, \xi_4)\} \quad \cdots \end{array}$$

Here, the constraint $\mathbf{Pub}(\mathbf{N}_{\gamma_1}(\xi_{0,o}, \xi_{0,c}))$ *comes from net!non, and the other constraints from* **check** *non* **is** *non.*$\cdots$. *By solving the qualifier constraints, we get* $\gamma_1 = \gamma'_1 = \mathbf{Pub}, \ldots$, *and*

are left with constraints on capability variables. By computing (an over-approximation of) the domain of each capability, we can reduce it to constraints on linear inequalities. For example, by letting $\xi_i = \{\mathbf{chk_{Pub}}(non, \xi_4) \mapsto x_i, \mathbf{end}(m) \mapsto y_i, \ldots\}$*, the last constraint is reduced to:*

$$x_7 \geq x_1 + x_2 + x_3 + 1 \quad y_7 \geq y_1 + y_2 + y_3 + 0 \quad \cdots$$

□

2.5 Implementation and Experiments

We have implemented a protocol verifier SpiCA2 based on the type system and inference algorithm discussed above[4]. We have tested SpiCA2 on several protocols with the results of the experiments shown in Table 2.1. Experiments were conducted using a machine with a 3GHz CPU and 2GB of memory.

The descriptions of the protocols used in the experiments are available at the above URL. `POSH`, `SOPH`, and `SOSH` are (spi$_{CA}$-notations of) the protocols given in Figure 2.1. `GNSL` is the generalized Needham-Schroeder-Lowe protocol [CM06] given in Section 2.8. `Otway-Ree` is Otway-Ree protocol using symmetric keys. `Iso-two-pass` is from [GJ04], and the remaining protocols are the Needham-Schroeder-Lowe protocol and its variants, taken from the sample programs of Cryptyc [HJ04] (but with type annotations and casts removed). `ns-flawed` is the original flawed version, `nsl-3` and `nsl-7` are 3- and 7-message versions of Lowe's fix, respectively. See [HJ04] for the other three. As the table shows, all the protocols have been correctly verified or rejected. Furthermore, verification succeeded in less than a second except for `GNSL`. For `GNSL`, the slow-down is caused by the explosion of the number of atomic capabilities to be considered, which blows up the number of linear inequalities obtained from capability constraints.

Protocols	Typing	Time (sec.)	Protocols	Typing	Time (sec.)
POSH	yes	0.001	ns-flawed	no	0.007
SOPH	yes	0.001	nsl-3	yes	0.015
SOSH	yes	0.001	nsl-7	yes	0.049
GNSL	yes	7.40	nsl-optimized	yes	0.012
Otway-Ree	yes	0.019	nsl-with-secret	yes	0.023
Iso-two-pass	yes	0.004	nsl-with-secret-optimized	yes	0.016

Table 2.1: Experimental results

2.6 Extensions

In this section we hint on how to modify our type system and inference algorithm to deal with other features. Formalization and implementation of the extensions are left for future work.

Our type system can be easily adopted to deal with non-injective correspondence [GJ02b], which allows multiple end-events to be matched by a single begin-event. It suffices to relax the typing rules, for example, by changing the rules for begin- and end-events to:

$$\frac{\Gamma; \varphi + \{\mathbf{end}(M) \mapsto r\} \vdash P \qquad r > 0}{\Gamma; \varphi \vdash \mathbf{begin}\, M.P} \qquad \frac{r > 0}{\Gamma; \varphi + \{\mathbf{end}(M) \mapsto r\} \vdash \mathbf{end}\, M}$$

[4]The implementation is mostly based on the formalization in the paper, except for a few extensions such as sum types and private channels to securely distribute initial keys. The implementation can be tested at http://www.kb.ecei.tohoku.ac.jp/~koba/spica2/.

The capabilities attached to a name can now be extracted without using the check operation:

$$\frac{\Gamma\ \varphi \vdash M : \mathbf{N}_\ell(\varphi_1, \varphi_2)}{\Gamma\ \varphi + \varphi_2 \vdash M : \mathbf{N}_\ell(\varphi_1, \varphi_2)}$$

Fournet et al. [FGM07] generalized begin- and end-events by allowing predicates to be defined by Datalog programs. For example, the process:

assume *employee*(a); **expect** *canRead(a, handbook)*

is safe in the presence of the clause "canRead(X,handbook) :- employee(X)". Here, the primitives **assume** and **expect** are like non-injective versions of **begin** and **end**. A similar type system may be obtained by extending our capabilities to mappings from ground atomic formulas to rational numbers (where $\varphi(L) > 0$ means L holds), and introducing rules for assume and expect similar to the rules above for begin and end-events. To handle clauses like "canRead(X,handbook) :- employee(X)" we may add the following rule:

$$\frac{\Gamma; \varphi + \{L \mapsto r\} \vdash P \qquad \begin{array}{c}\text{There is an (instance of) clause } L :- L_1, \ldots, L_k \\ r \leq \varphi(L_i) \text{ for each } i \in \{1, \ldots, k\}\end{array}}{\Gamma; \varphi \vdash P}$$

which would allow us to derive a capability for L whenever there are capabilities for $L_1, \ldots, L_k$. To reduce capability constraints to linear programming problems the algorithm could be extended to obtain the domain of each effect, taking clauses into account (i.e. if there is a clause $L :- L_1, \ldots, L_k$ and $\theta L_1, \ldots, \theta L_k$ are in the domain of φ, we add θL to the domain of φ).

To deal with trust and witness in [GJ04] we need to mix type environments and capabilities so that type environments can also be attached to names and passed around.

2.7 Related Work

The present work extends two lines of previous work: Gordon and Jeffrey's type systems for authenticity [GJ03, GJ04], and Kikuchi and Kobayashi's work to enable type inference for symmetric cryptographic protocols [KK09]. In our opinion the extension is non-trivial, requiring the generalization of name types and a redesign of the type system. This has yielded a fully-automated and efficient protocol verifier. As for the expressive power, the fragment of Gordon and Jeffrey's type system (subject to minor restrictions) without trust and witness can be easily embedded into our type system. On the other hand, thanks to the uniform treatment of name types in terms of capabilities and obligations, our type system can express protocols that are not typable in Gordon and Jeffrey's type system, like the GNSL multi-party protocol [CM06].

Gordon et al. [BBF+08, BFG10] extended their work to verify source code-level implementation of cryptographic protocols by using refinement types. Their type systems still require refinement type annotations. We plan to extend the ideas of the present work to enable partial type inference for their type system. Bugliesi, Focardi, and Maffei [BFM05, FMP05, BFM07] have proposed a protocol verification method that is closely related to Gordon and Jeffrey's type systems. They [FMP05] developed an algorithm for automatically inferring *tags* (which roughly correspond to Gordon and Jeffrey's types in [GJ03, GJ04]). Their inference algorithm is based on exhaustive search of taggings by backtracking, hence our type inference would be more efficient. As in Gordon and Jeffrey type system, their tagging and typing system is specialized for the typical usage of nonces in two-party protocols, and appears to be inapplicable to multi-party protocols like GNSL.

There are automated protocol verification tools based on other approaches, such as ProVerif [Bla02] and Scyther [Cre08]. Advantages of our type-based approach are: (i) it allows modular

verification of protocols[5]; (ii) it sets up a basis for studies of partial or full type inference for more advanced type systems for protocol verification [BFG10] (recall Section 2.6); and (iii) upon successful verification, it generates types as a certificate, which explains why the protocol is safe, and can be independently checked by other type-based verifiers [GJ04, BFG10]. On the other hand, ProVerif [Bla02] and Scyther [Cre08] have an advantage that they can generate an attack scenario given a flawed protocol. Thus, we think that our type-based tool is complementary to existing tools.

2.8 Relation to Gordon-Jeffrey Type System

We now turn to the subject of relating our type system to that of Gordon and Jeffrey. There are two main points here. First, we show that the fragment of the Gordon-Jeffrey type system without witness and trust can be embedded into our type system. We make some additional restrictions regarding nonce types but these appear to be without loss of expressive power for practice purposes. Second, we show that our formulation of nonce types actually allows us to type realistic protocols untypable in the Gordon-Jeffrey type system.

2.8.1 Partial Embedding of Gordon and Jeffrey's Type System

Restrictions

In order to show an embedding of their type system into ours we have to make a few modifications. Most notably, we (i) leave out an embedding for witness and trust processes, (ii) inline the message subsumption rule, (iii) modify check atomic effects to additionally contain an effect es, and (iv) change the typing of processes dealing with nonce types.

Modification (ii) means that the subsumption rule for messages is removed and inlined in rules Proc Output Un and Proc Input Un (and similarly for Proc Repeat Input Un) instead:

$$\frac{\Gamma \vdash_{\mathbf{GJ}} M : \mathsf{Un} \qquad \Gamma \vdash_{\mathbf{GJ}} N : T \qquad T \leq_{\mathbf{GJ}} \mathsf{Un}}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{out}\ M\ N : []} \quad \text{(Proc Output Un)}$$

$$\frac{\Gamma \vdash_{\mathbf{GJ}} M : \mathsf{Un} \qquad \Gamma, y : T \vdash_{\mathbf{GJ}} P : es \qquad \mathsf{Un} \leq_{\mathbf{GJ}} T}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{inp}\ M\ (y : T); P : es} \quad \text{(Proc Input Un)}$$

This modification is justified by the belief that honest processes should not have to apply subsumptions in more general ways than this, in that doing so means changing a type from or to something else than Un.

Modifications (iii) and (iv) mean that typing rules Proc Challenge and Proc Check are changed as follows:

$$\frac{\Gamma \vdash_{\mathbf{GJ}} fs \qquad \Gamma, x : l\ \mathsf{Challenge}\ fs \vdash_{\mathbf{GJ}} P : es}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{new}\ (x : l\ \mathsf{Challenge}\ fs); P : es - [\mathsf{check}\ l\ x\ fs]} \quad \text{(Proc Challenge)}$$

[5] Although the current implementation of SpiCA2 only supports whole protocol analysis, it is easy to extend it to support partial type annotations to enable modular verification. For that purpose, it suffices to allow bound variables to be annotated with types, and generate the corresponding constraints during type inference. For example, for a type-annotated input $M?(x : \tau_1).P$, we just need to add the subtype constraint $\tau_1 \leq \tau$ to rule T-In.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbf{GJ}} M : l\ \mathsf{Challenge}\ es_C \\ \Gamma \vdash_{\mathbf{GJ}} N : l\ \mathsf{Response}\ es_R \qquad \Gamma \vdash_{\mathbf{GJ}} P : fs \\ es = fs - (es_C + es_R) \qquad es'_C = es_C\end{array}}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{check}\ M\ \mathsf{is}\ N; P : es + [\mathsf{check}\ l\ M\ es'_C]} \quad \text{(Proc Check)}$$

One consequence of this is that M in rule Proc Check can no longer be bound to names with different check capabilities. Moreover, while the addition of condition $es'_C = es_C$ makes rule Proc Check more restrictive than in the original formulation, breaking this condition does require use of either subtyping or matching in a way that respectively should not be done by honest processes, or does not appear to be required by a significant number of protocols. In the former case, subtyping must be used to turn a public nonce into a private nonce. In the latter case, match can be used to turn a check capability for one name into a check capability for another name. This however, seems to be possible only for protocols that deadlock.

We apply a few less important modifications to the type system as well. Type Top is removed and typing rules Proc Begin and Proc End are modified to simply require M to be of some type T instead:

$$\frac{\Gamma \vdash_{\mathbf{GJ}} M : T \qquad \Gamma \vdash_{\mathbf{GJ}} P : es}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{begin}\ M; P : es - [\mathsf{end}\ M]} \quad \text{(Proc Begin)}$$

$$\frac{\Gamma \vdash_{\mathbf{GJ}} M : T \qquad \Gamma \vdash_{\mathbf{GJ}} P : es}{\Gamma \vdash_{\mathbf{GJ}} \mathsf{end}\ M; P : es + [\mathsf{end}\ M]} \quad \text{(Proc End)}$$

As far as we checked, all the protocols (without trust and witness) typed in [GJ04] are typable under all the modifications above.

For the calculus we first consider a variant of the process calculus without $\mathsf{inl}(M)$ and $\mathsf{inr}(M)$ messages, and without case processes. Secondly, we restrict end processes to match our non-continuous variant. Thirdly, we restrict the generation of key pairs so that messages $\mathsf{Encrypt}(M)$ and $\mathsf{Decrypt}(M)$ may only occur immediately following the creation of a key pair (see below). The restrictions on the calculus can be removed by an easy extension of our calculus and type system, and are improsed here just for the sake of simplicity.

Embedding

To ease the presentation we first add a derived process to both calculi

$$\mathbf{let}\ y\ \mathbf{is}\ x\ \mathbf{in}\ P = \mathbf{match}\ (c, x)\ \mathbf{is}\ (c, y).P$$

for some constant c, along with typing rule

$$\frac{\Gamma; \varphi_1 \vdash x : \tau \qquad \Gamma, y : \tau; \varphi_2 \vdash P}{\Gamma; \varphi_1 + \varphi_2 \vdash \mathbf{let}\ y\ \mathbf{is}\ x\ \mathbf{in}\ P} \quad \text{(T-Let)}$$

derivable from typing rules T-Match and T-Pair.

The central ingredient in the embedding is the mapping of types. For this we first have a straightforward mapping of messages M and effects es relying purely on syntactical conversion; for this reason we shall often simply write M instead of $[M]$ and es instead of $[es]$. We then define the mapping $[T]$ of a Gordon-Jeffrey type T as in Figure 2.10. We extend this to environments Γ in the point-wise manner. Note that as discussed above we consider a variant of the Gordon-Jeffrey type system withtop Top and Sum types and our mapping is left undefined

$$
\begin{array}{rcl}
[(x{:}T, U)] & = & [T] \times [0/x][U] \\
[\mathsf{Un}] & = & \mathbf{N_{Pub}}(\emptyset, \emptyset) \\
[\mathsf{SharedKey}(T)] & = & \mathbf{SKey}([T]) \\
[\mathsf{Public\ Challenge}\ es] & = & \mathbf{N_{Pub}}([es], \emptyset) \\
[\mathsf{Private\ Challenge}\ es] & = & \mathbf{N_{Pr}}([es], \emptyset) \\
[\mathsf{Public\ Responce}\ es] & = & \mathbf{N_{Pub}}(\emptyset, [es]) \\
[\mathsf{Private\ Responce}\ es] & = & \mathbf{N_{Pr}}(\emptyset, [es]) \\
[\mathsf{Encrypt\ Key}(T)] & = & \mathbf{EKey}([T]) \\
[\mathsf{Decrypt\ Key}(T)] & = & \mathbf{DKey}([T])
\end{array}
$$

Figure 2.10: Type mapping

$$
\begin{array}{rcl}
[\mathsf{cast}\ x\ \mathsf{is}\ (y{:}T); P] & = & \mathbf{let}\ y\ \mathbf{is}\ x\ \mathbf{in}\ [P] \\
[\mathsf{check}\ x\ \mathsf{is}\ (y{:}T); P] & = & \mathbf{check}\ x\ \mathbf{is}\ y.[P] \\
[\mathsf{end}\ M; 0] & = & \mathbf{end}\ M \\
[\mathsf{begin}\ M; P] & = & \mathbf{begin}\ M.[P] \\
[\mathsf{new}\ (x{:}\mathsf{Un}); P] & = & (\nu x)[P] \\
[\mathsf{new}\ (x{:}l\ \mathsf{Challenge}\ es); P] & = & (\nu x)[P] \\
[\mathsf{new}\ (x{:}\mathsf{SharedKey}(T)); P] & = & (\nu_{sym} x)[P] \\
\left[\begin{array}{l}\mathsf{new}\ (x{:}\mathsf{KeyPair}(T)); \\ \mathsf{let}\ y\ \mathsf{is}\ \mathsf{Encrypt}(x)\ \mathsf{in} \\ \mathsf{let}\ z\ \mathsf{is}\ \mathsf{Decrypt}(x)\ \mathsf{in} \\ P \quad (x \notin \mathbf{FN}(P))\end{array}\right] & = & (\nu_{asym} y, z)[P] \\
[\mathsf{new}\ (x{:}\mathsf{Un}); P] & = & (\nu x)[P] \\
[\mathsf{out}\ M\ N] & = & M!N \\
[\mathsf{inp}\ M\ (x{:}T); P] & = & M?x.[P] \\
[\mathsf{repeat\ inp}\ M\ (x{:}T); P] & = & *M?x.[P] \\
[\mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U); P] & = & \mathbf{split}\ M\ \mathbf{is}\ (x, y).[P] \\
[\mathsf{match}\ M\ \mathsf{is}\ (N, y{:}U); P] & = & \mathbf{match}\ M\ \mathbf{is}\ (N, y).[P] \\
[\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P] & = & \mathbf{decrypt}\ M\ \mathbf{is}\ \{x\}_N.[P] \\
[\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}T|\!\}_{N^{-1}}; P] & = & \mathbf{decrypt}\ M\ \mathbf{is}\ \{\!|x|\!\}_{N^{-1}}.[P] \\
[P \mid Q] & = & [P] \mid [Q] \\
[\mathsf{stop}] & = & \mathbf{0}
\end{array}
$$

Figure 2.11: Process mapping

for these. As we furthermore allow only a restricted use of KeyPair types the mapping is also left undefined for these as well as for CR types since these should not occur in user code.

In the mapping of processes (Figure 2.11) we use the provided typing information in the case of name restriction. As discussed above we impose some restrictions on processes and the mapping is left undefined for these; for the remaining processes the mapping is defined recursively.

Lemma 2.8.1. *If* $\Gamma, x : T \vdash_{\mathbf{GJ}} P$ *and* $x \notin fn(P)$ *then* $\Gamma \vdash_{\mathbf{GJ}} P$.

Proof. Follows from Lemma 10 in the technical report for the Gordon-Jeffrey type system [GJ02a] □

Lemma 2.8.2. *If* $Public(T)$ *then* $\mathbf{Pub}([T])$. *If* $Tainted(T)$ *then* $\mathbf{Taint}([T])$.

Proof. By straightforward induction in the derivation of $Public(T)$ and $Tainted(T)$ using their algorithmic formulation. Rules Tainted Top, Public Sum, Tainted Sum, Public Keypair, Tainted Keypair, and Public CR are not considered. □

Lemma 2.8.3. *If* $T \leq_{\mathbf{GJ}} \mathsf{Un}$ *then* $\mathbf{Pub}([T])$. *If* $\mathsf{Un} \leq_{\mathbf{GJ}} T$ *then* $\mathbf{Taint}([T])$.

Proof. In both cases we see that rule Sub Public Tainted must have been used to derive the subtyping expression. In both cases Lemma 2.8.2 gives us the desired result. □

Lemma 2.8.4. *If* $\Gamma \vdash_{\mathbf{GJ}} M : T$ *then* $[\Gamma]; \emptyset \vdash [M] : [T]$.

Proof. By straightforward induction in the derivation of $\Gamma \vdash_{\mathbf{GJ}} M : T$. Note that rules Msg Subsum, Msg Inl, Msg Inr, and Msg Part cannot happen by restriction. □

Theorem 2.8.5. *If* $\Gamma \vdash_{\mathbf{GJ}} P : es$ *then* $[\Gamma]; [es] \vdash [P]$.

Proof. By induction in the derivation of $\Gamma \vdash_{\mathbf{GJ}} P : es$.

- Case Proc Subsum: by induction hypothesis and the fact that $es \leq es + fs$, we can apply rule T-CSub to obtain the desired result.
- Case Proc Output Un: since $\Gamma \vdash_{\mathbf{GJ}} M : \mathsf{Un}$ we have by Lemma 2.8.4 that $[\Gamma]; \emptyset \vdash M : [\mathsf{Un}]$. As $[\mathsf{Un}] = \mathbf{N_{Pub}}(\emptyset, \emptyset)$ we get that the first condition for rule T-Out is satisfied. Since $\Gamma \vdash_{\mathbf{GJ}} N : T$ we can again apply Lemma 2.8.4 to obtain $[\Gamma]; \emptyset \vdash N : [T]$ thereby satisfying the second condition for rule T-Out. Finally, since $T \leq_{\mathbf{GJ}} \mathsf{Un}$ we get from Lemma 2.8.3 that $\mathbf{Pub}([T])$ and can then satisfy the final condition of rule T-Out.
- Case Proc Input Un: since $\Gamma \vdash_{\mathbf{GJ}} M : \mathsf{Un}$ we have by Lemma 2.8.4 that $[\Gamma]; \emptyset \vdash M : [\mathsf{Un}]$. As $[\mathsf{Un}] = \mathbf{N_{Pub}}(\emptyset, \emptyset)$ we get that the first condition for rule T-In is satisfied. Since $\Gamma, y : T \vdash_{\mathbf{GJ}} P : es$ we have by induction hypothesis that the second condition is satisfied. Finally, since $\mathsf{Un} \leq_{\mathbf{GJ}} T$ we get from Lemma 2.8.3 that $\mathbf{Taint}([T])$ and can then satisfy the final condition of rule T-In.
- Case Proc Repeat Input Un: similar to case Proc Output Un but also using rule T-Rep.
- Case Proc Par: by the induction hypothesis we can immediately apply rule T-Par.
- Case Proc Res: we treat the different cases of T separately:
  - $T = \mathsf{Un}$: since $[\mathsf{Un}] = \mathbf{N_{Pub}}(\emptyset, \emptyset)$ we can apply the induction hypothesis and rule T-NewN to obtain the desired result.
  - $T = \mathsf{SharedKey}(T')$: by the fact that $\mathbf{SKey}([T']) = [\mathsf{SharedKey}(T')]$ we can apply the induction hypothesis and rule T-NewSk to obtain the desired result.
  - $T = \mathsf{KeyPair}(T')$: by the restricted used of KeyPair we know that two next constructions in P are **let** constructs follows by a process P'. By the typing of P we get that $E, x : \mathsf{KeyPair}(T'), y : \mathsf{Encrypt\ Key}(T'), z : \mathsf{Encrypt\ Key}(T') \vdash_{\mathbf{GJ}} P' : es$. By the fact that $x \notin fn(P')$ Lemma 2.8.1 gives us that $E, y : \mathsf{Encrypt\ Key}(T'), z : \mathsf{Encrypt\ Key}(T') \vdash_{\mathbf{GJ}} P' : es$. By induction hypothesis $[E], y : [\mathsf{Encrypt\ Key}(T')], z : [\mathsf{Encrypt\ Key}(T')]; [es] \vdash [P']$. Since $[\mathsf{Encrypt\ Key}(T')] = \mathbf{EKey}([T'])$ and $[\mathsf{Decrypt\ Key}(T')] = \mathbf{DKey}([T'])$ we can apply rule T-NewAk to obtain the desired result.

- Case PROC SPLIT: follows by the induction hypothesis, Lemma 2.8.4, and rule T-SPLIT.
- Case PROC MATCH: follows by the induction hypothesis, Lemma 2.8.4, and rule T-MATCH.
- Case PROC CASE: cannot happen by our restrictions.
- Case PROC SYMM: follows by the induction hypothesis, Lemma 2.8.4, and rule T-SDEC.
- Case PROC ASYMM: follows by the induction hypothesis, Lemma 2.8.4, and rule T-ADEC.
- Case PROC BEGIN: follows by the induction hypothesis and rule T-BEGIN; if es does not contain an $\mathbf{end}(M)$ we have to extend it first using T-CSUB.
- Case PROC END: by our restrictions $P = \mathsf{end}\, L; \mathsf{stop}$ and hence the results follows by rule T-END.
- Case PROC WITNESS: cannot happen by our restrictions.
- Case PROC TRUST: cannot happen by our restrictions.
- Case PROC CAST: by assumption we have $\Gamma \vdash_{\mathbf{GJ}} x : l\ \mathsf{Challenge}\ es_C$ and also $\Gamma, x : l\ \mathsf{Response}\ es_R \vdash_{\mathbf{GJ}} P : fs$. Lemma 2.8.4 then gives us that $[\Gamma]; \emptyset \vdash x : [l\ \mathsf{Challenge}\ es_C]$ and the induction hypothesis that $[\Gamma], x : [l\ \mathsf{Response}\ es_R]; [fs] \vdash [P]$. Since $[l\ \mathsf{Challenge}\ es_C] = \mathbf{N}_l([es_C], \emptyset)$ we can apply rule T-NAME to obtain $[\Gamma]; [es_C] + [es_R] \vdash x : \mathbf{N}_l(\emptyset, [es_R])$. As $\mathbf{N}_l(\emptyset, [es_R]) = [l\ \mathsf{Response}\ es_R]$ we can apply rule T-LET to obtain the desired result.
- Case PROC CHECK: by assumption we have $\Gamma \vdash_{\mathbf{GJ}} M : l\ \mathsf{Challenge}\ es_C$, $\Gamma \vdash_{\mathbf{GJ}} N : l\ \mathsf{Response}\ es_R$, and $\Gamma \vdash_{\mathbf{GJ}} P : fs$. Lemma 2.8.4 then gives us that $[\Gamma]; \emptyset \vdash M : [l\ \mathsf{Challenge}\ es_C]$ and $[\Gamma]; \emptyset \vdash N : [l\ \mathsf{Response}\ es_C]$. Since $[l\ \mathsf{Challenge}\ es_C] = \mathbf{N}_l([es_C], \emptyset)$ and $[l\ \mathsf{Response}\ es_R] = \mathbf{N}_l(\emptyset, [es_R])$ we can satisfy the two first premises of rule T-CHK using $\varphi_1 = \varphi_2 = \emptyset$. Now let $\varphi_3 = [es] = [fs] - ([es_C] + [es_R])$, $\varphi_4 = [es_C]$, and $\varphi_5 = [es_R]$. Since $\Gamma \vdash_{\mathbf{GJ}} P : fs$ we have by the induction hypothesis that $[\Gamma]; \varphi_3 + \varphi_4 + \varphi_5 \vdash [P]$ and we can finally apply rule T-CHK.
- Case PROC CHALLENGE: follows by induction hypothesis and rule T-RES.

□

2.8.2 Limitations of Gordon and Jeffrey Type System

The converse of the result of the previous subsection does not hold, i.e. there are some realistic protocols that are typable in our type system but not in the Gordon-Jeffrey type system. This is a consequence of how nonces are typed: in their type system, nonce types are given two kinds of types: $\ell\ \mathsf{Challenge}\ es$ and $\ell\ \mathsf{Response}\ es$. This forces each nonce to be used in at most two phases, first as a challenge, and then as a response. Our name types do not impose such restriction. The rest of this section illustrates two cases of protocols typable in our type system but not in Gordon and Jeffrey's type system.

Generalised Needham-Schroeder-Lowe

The GNSL multi-party protocol [CM06] establishes mutual authentication between p parties using a minimal number of messages. For $p = 3$ with participants named R_0, R_1, and R_2, the protocol looks as follows:

```
R0 -> R1: {|R0,R2,n0|}pk1
R1 -> R2: {|R0,R1,n0,n1|}pk2
R2 -> R0: {|R1,R2,n0,n1,n2|}pk0
R0 -> R1: {|n1,n2|}pk1
R1 -> R2: {|n2|}pk2
```

where n_i is a nonce generated by R_i and pk_i the public key of a key pair belonging to R_i.

Participant R_0 first sends his nonce n_0 to R_1 who appends his nonce n_1 before forwarding to R_2. Likewise, R_2 appends his nonce n_2 before sending all nonces back to R_0. For the second round, R_0 checks his nonce against the one received from R_2 and sends n_1 and n_2 to R_1. After checking his nonce, R_1 sends n_2 to R_2 who then also checks his nonce.

The authenticity property dictates that each party agrees with both of the other parties on who the participants are, and is specified like so:

```
R0 -> R1: {|R0,R2,n0|}pk1
R1 begins (R0,R1,R2,01)
R1 begins (R0,R1,R2,21)
R1 -> R2: {|R0,R1,n0,n1|}pk2
R2 begins (R0,R1,R2,02)
R2 begins (R0,R1,R2,12)
R2 -> R0: {|R1,R2,n0,n1,n2|}pk0
R0 ends   (R0,R1,R2,01)
R0 ends   (R0,R1,R2,02)
R0 begins (R0,R1,R2,10)
R0 begins (R0,R1,R2,20)
R0 -> R1: {|n1,n2|}pk1
R1 ends   (R0,R1,R2,10)
R1 ends   (R0,R1,R2,12)
R1 -> R2: {|n2|}pk2
R2 ends   (R0,R1,R2,20)
R2 ends   (R0,R1,R2,21)
```

for some constants $01, \ldots, 21$. Note that for this property to hold we must assume that none of the parties R_0, R_1, and R_2 are compromised.

From the type system's point of view, the authenticity property e.g. means that a end-capability from both R_1 and R_2 must be transferred to R_0 using one nonce n_0. This is a problem for Gordon and Jeffrey's type system since capabilities can only be attached to nonces once due to the fact that the Proc Cast typing rule will only accept a Challenge type and additionally turn it into a Response type. Our type system does not have this limitation and can type the protocol with the following initial types for the nonces:

$$n_0 : \mathbf{N_{Pr}}(\{\mathbf{end}(\ldots, 01), \mathbf{end}(\ldots, 21), \mathbf{end}(\ldots, 02)\}, \emptyset)$$
$$n_1 : \mathbf{N_{Pr}}(\{\mathbf{end}(\ldots, 10), \mathbf{end}(\ldots, 12)\}, \emptyset)$$
$$n_2 : \mathbf{N_{Pr}}(\{\mathbf{end}(\ldots, 20), \mathbf{end}(\ldots, 21)\}, \emptyset)$$

so that the type of n_0 is later changed by R_1 to $\mathbf{N_{Pr}}(\{\mathbf{end}(\ldots, 02)\}, \emptyset)$ and then by R_2 to $\mathbf{N_{Pr}}(\emptyset, \emptyset)$. When n_0 makes it back to R_0 it can extract capabilities

$$\{\mathbf{end}(\ldots, 01), \mathbf{end}(\ldots, 21), \mathbf{end}(\ldots, 02)\}$$

and use $\mathbf{end}(\ldots, 21)$ to discharge the obligation attached to n_2. These changes of name types cannot be expressed in Gordon and Jeffrey's type system.

SOPH Handshakes

Another example of a protocol that is typable in our type system but not in Gordon and Jeffrey's is the SOPH handshake protocol in Figure 2.1. As mentioned in Example 2.3.1 in Section 2.3.3, pk_B should have type $\mathbf{EKey}(\mathbf{Un} \times \mathbf{N_{Pub}}(\{\mathbf{end}(0) \mapsto 1\}, \emptyset))$, which corresponds to $\mathsf{Encrypt\ Key}(x : \mathsf{Un}, \mathsf{Pub\ Challenge}\ [\mathbf{end}(x)])$ in Gordon and Jeffrey's type system. The key

$$
\begin{array}{l}
[\![y]\!]_x = x!y \\
[\![(M_1, M_2)]\!]_x = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.x!(z_1, z_2)) \\
[\![\{M_1\}_{M_2}]\!]_x = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.x!\{z_1\}_{z_2}) \\
[\![\{\!|M_1|\!\}_{M_2}]\!]_x = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.x!\{\!|z_1|\!\}_{z_2}) \\
\\
[\![\mathbf{0}]\!] = \mathbf{0} \\
[\![M_1!M_2]\!] = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.z_1!z_2) \\
[\![M_1?x.P]\!] = (\nu y_1)([\![M_1]\!]_{y_1} \mid y_1?z_1.z_1?x.[\![P]\!]) \\
[\![*P]\!] = *[\![P]\!] \\
[\![(\nu x)P]\!] = (\nu x)[\![P]\!] \\
[\![(\nu_{sym} x)P]\!] = (\nu_{sym} x)[\![P]\!] \\
[\![(\nu_{asym} x_1, x_2)P]\!] = (\nu_{asym} x_1, x_2)[\![P]\!] \\
\\
[\![\mathbf{check}\ M_1\ \mathbf{is}\ M_2.P]\!] = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.\mathbf{check}\ z_1\ \mathbf{is}\ z_2.[\![P]\!]) \\
[\![\mathbf{split}\ M\ \mathbf{is}\ (x_1, x_2).P]\!] = (\nu y)([\![M]\!]_y \mid y?z.\mathbf{split}\ z\ \mathbf{is}\ (x_1, x_2).[\![P]\!]) \\
[\![\mathbf{match}\ M_1\ \mathbf{is}\ (M_2, x).P]\!] = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.\mathbf{match}\ z_1\ \mathbf{is}\ (z_2, x).[\![P]\!]) \\
[\![\mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P]\!] = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.\mathbf{decrypt}\ z_1\ \mathbf{is}\ \{x\}_{z_2}.[\![P]\!]) \\
[\![\mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2{}^{-1}}.P]\!] = (\nu y_1)(\nu y_2)([\![M_1]\!]_{y_1} \mid [\![M_2]\!]_{y_2} \mid y_1?z_1.y_2?z_2.\mathbf{decrypt}\ z_1\ \mathbf{is}\ \{\!|x|\!\}_{z_2{}^{-1}}.[\![P]\!]) \\
[\![\mathbf{begin}\, M.P]\!] = \mathbf{begin}\, M.[\![P]\!] \\
[\![\mathbf{end}\, M]\!] = \mathbf{end}\, M
\end{array}
$$

Figure 2.12: Encoding of Messages and Processes

pk_B is public, but the **Pub** predicate does not hold for this type in Gordon and Jeffrey's type system [GJ04].[6] The discrepancy comes from the fact that $\mathbf{Taint}(\mathbf{N_{Pub}}(\varphi, \emptyset))$ holds for arbitrary φ in our type system, but the corresponding condition $\mathbf{Taint}(\mathsf{Pub\ Challenge}\ \varphi)$ holds only for the case $\varphi = \emptyset$ in Gordon and Jeffrey's. This seems to be caused by the difference in the rules for typing check operations as discussed in the previous subsection. Because of the difference, allowing $\mathbf{Taint}(\mathsf{Pub\ Challenge}\ \varphi)$ to hold for arbitrary φ is unsound for Gordon and Jeffrey's type system.

2.9 Proofs of Lemmas

We here give proofs of the lemmas used for the soundness theorem.

2.9.1 Proof of Lemma 2.3.1

In Figure 2.12 we define an encoding of messages and processes used for the lemma: $[\![M]\!]_x$ translates a message M (that may not be well-typed) to a well-typed process that sends the value of M on channel x, and $[\![P]\!]$ translates a process P to an equivalent, well-typed process. We assume below that renaming is applied as necessary to avoid the name clashing.

By straightforward induction on the structures of M and P we can prove $y_1 : \mathbf{Un}, \ldots, y_n : \mathbf{Un}, x : \mathbf{Un}; \emptyset \vdash [\![M]\!]_x : \mathbf{Un}$ and $z_1 : \mathbf{Un}, \ldots, z_m : \mathbf{Un}; \emptyset \vdash [\![P]\!]$ where $\mathbf{FN}(M) = \{y_1, \ldots, y_n\}$ and $\mathbf{FN}(P) = \{z_1, \ldots, z_m\}$. It is also obvious that for any reduction sequence of $P \mid Q$ there is a corresponding for $[\![P]\!] \mid Q$. Thus, the required result of the lemma holds for $O' = [\![O]\!]$.

[6] Confirmed by email discussion with Gordon and Jeffrey.

$$\Gamma, x : \tau; \varphi \vdash_{\mathbf{ex}} x : \tau \qquad \text{(ExT-Var)}$$

$$\Gamma, n : \mathbf{N}_\ell(\_,\_); \varphi \vdash_{\mathbf{ex}} n^{(\varphi_1,\varphi_2)} : \mathbf{N}_\ell(\varphi_1,\varphi_2) \qquad \text{(ExT-Name)}$$

$$\frac{\Gamma; \varphi \vdash_{\mathbf{ex}} V : \mathbf{N}_\ell(\varphi_1,\varphi_2)}{\Gamma; \varphi + \varphi_1' + \varphi_2' \vdash_{\mathbf{ex}} \mathbf{addC}(V,\varphi_1',\varphi_2') : \mathbf{N}_\ell(\varphi_1 - \varphi_1', \varphi_2 + \varphi_2')} \qquad \text{(ExT-AddC)}$$

$$\frac{\Gamma; \varphi_1 \vdash_{\mathbf{ex}} M_1 : \tau_1 \qquad \Gamma; \varphi_2 \vdash_{\mathbf{ex}} M_2 : [M_1/0]\tau_2}{\Gamma; \varphi_1 + \varphi_2 \vdash_{\mathbf{ex}} (M_1, M_2) : \tau_1 \times \tau_2} \qquad \text{(ExT-Pair)}$$

$$\frac{\Gamma; \varphi_1 \vdash_{\mathbf{ex}} M_1 : \tau_1 \qquad \Gamma; \varphi_2 \vdash_{\mathbf{ex}} M_2 : \mathbf{SKey}(\tau_1)}{\Gamma; \varphi_1 + \varphi_2 \vdash_{\mathbf{ex}} \{M_1\}_{M_2} : \mathbf{N}_\ell(\emptyset,\emptyset)} \qquad \text{(ExT-SEnc)}$$

$$\frac{\Gamma; \varphi_1 \vdash_{\mathbf{ex}} M_1 : \tau \qquad \Gamma; \varphi_2 \vdash_{\mathbf{ex}} M_2 : \mathbf{EKey}(\tau)}{\Gamma; \varphi_1 + \varphi_2 \vdash_{\mathbf{ex}} \{\!|M_1|\!\}_{M_2} : \mathbf{N}_\ell(\emptyset,\emptyset)} \qquad \text{(ExT-AEnc)}$$

$$\frac{\Gamma; \varphi \vdash_{\mathbf{ex}} M : \tau' \qquad \tau' \leq_{\mathbf{ex}} \tau}{\Gamma; \varphi \vdash_{\mathbf{ex}} M : \tau} \qquad \text{(ExT-Sub)}$$

Figure 2.13: Typing Rules for Extended Messages

2.9.2 Proof of Lemma 2.3.2

To prove Lemma 2.3.2 we extend the syntax of processes and the typing rules in order to express invariants preserved by reductions.

Extended Processes

We extend the syntax of processes in order to make it explicit what obligations and capabilities are carried by each name, and when they are attached to the name. Below we distinguish between (bound) variables, ranged over by x, and (free) names, ranged over by n. The sets of *extended messages and processes* are given by:

$$\begin{aligned}
&M(\text{ext. messages}) ::= v \mid \mathbf{addC}(M,\varphi_1,\varphi_2) \mid (M_1,M_2) \mid \{M_1\}_{M_2} \mid \{\!|M_1|\!\}_{M_2}\\
&V(\text{values}) ::= v \mid (V_1,V_2) \mid \{V_1\}_{V_2} \mid \{\!|V_1|\!\}_{V_2}\\
&v ::= x \mid n^{(\varphi_1,\varphi_2)}\\
&P(\text{ext. processes}) ::= \mathbf{0} \mid M_1!M_2 \mid M?x.P \mid (P_1 \mid P_2) \mid *P\\
&\qquad\qquad \mid (\nu x:\tau)P \mid (\nu_{sym} k:\tau)P \mid (\nu_{asym} k_1:\tau_1, k_2:\tau_2)P\\
&\qquad\qquad \mid \mathbf{check}\ M_1\ \mathbf{is}\ M_2.P\\
&\qquad\qquad \mid \mathbf{split}\ M\ \mathbf{is}\ (x,y).P \mid \mathbf{match}\ M_1\ \mathbf{is}\ (M_2,y).P\\
&\qquad\qquad \mid \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P \mid \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2{}^{-1}}.P\\
&\qquad\qquad \mid \mathbf{begin}\, V.P \mid \mathbf{end}\, V
\end{aligned}$$

and the typing rules for extended processes are shown in Figure 2.13 and 2.14. In Figure 2.13, $\leq_{\mathbf{ex}}$ is the least reflexive relation that satisfies the following rules:

$$\frac{\mathbf{Pub}(\tau) \qquad \mathbf{Taint}(\tau')}{\tau \leq_{\mathbf{ex}} \tau'} \qquad \text{(ExSubT-PubTaint)}$$

$$\frac{\varphi_1 \leq \varphi_1' \qquad \varphi_2 \geq \varphi_2'}{\mathbf{N}_\ell(\varphi_1,\varphi_2) \leq_{\mathbf{ex}} \mathbf{N}_\ell(\varphi_1',\varphi_2')} \qquad \text{(ExSubT-Name)}$$

Lemma 2.9.1. *If* $\tau_1 \leq_{\mathbf{ex}} \tau_2$ *and* $\tau_2 \leq_{\mathbf{ex}} \tau_3$ *then* $\tau_1 \leq_{\mathbf{ex}} \tau_3$.

$$\Gamma;\emptyset \vdash_{\mathbf{ex}} \mathbf{0}$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M_1 : \mathbf{N}_\ell(\emptyset,\emptyset) \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} M_2 : \tau \qquad \mathbf{Pub}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash_{\mathbf{ex}} M_1!M_2} \quad (\text{ExT-Out})$$

$$\frac{\Gamma_1;\varphi_1 \vdash_{\mathbf{ex}} M : \mathbf{N}_\ell(\emptyset,\emptyset) \qquad \Gamma_2, x:\tau;\varphi_2 \vdash_{\mathbf{ex}} P \qquad \mathbf{Taint}(\tau)}{\Gamma;\varphi_1+\varphi_2 \vdash_{\mathbf{ex}} M?x.P} \quad (\text{ExT-In})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} P_1 \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} P_2}{\Gamma;\varphi_1+\varphi_2 \vdash_{\mathbf{ex}} P_1 \mid P_2} \quad (\text{ExT-Par})$$

$$\frac{\Gamma;\emptyset \vdash_{\mathbf{ex}} P}{\Gamma;\emptyset \vdash_{\mathbf{ex}} *P} \quad (\text{ExT-Rep})$$

$$\frac{\Gamma, x:\mathbf{N}_\ell(\varphi_1,\emptyset);\varphi+\{\mathbf{chk}_\ell(x,\varphi_1)\mapsto 1\} \vdash_{\mathbf{ex}} P}{\Gamma;\varphi \vdash_{\mathbf{ex}} (\nu x:\mathbf{N}_\ell(\varphi_1,\emptyset))P} \quad (\text{ExT-NewN})$$

$$\frac{\Gamma, x:\mathbf{SKey}(\tau);\varphi \vdash_{\mathbf{ex}} P}{\Gamma;\varphi \vdash_{\mathbf{ex}} (\nu_{sym} x:\mathbf{SKey}(\tau))P} \quad (\text{ExT-NewSk})$$

$$\frac{\Gamma, k_1:\mathbf{EKey}(\tau), k_2:\mathbf{DKey}(\tau);\varphi \vdash_{\mathbf{ex}} P}{\Gamma;\varphi \vdash_{\mathbf{ex}} (\nu_{asym} k_1:\mathbf{EKey}(\tau), k_2:\mathbf{DKey}(\tau))P} \quad (\text{ExT-NewAk})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} M_2 : \mathbf{SKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash_{\mathbf{ex}} P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash_{\mathbf{ex}} \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{x\}_{M_2}.P} \quad (\text{ExT-SDec})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} M_2 : \mathbf{DKey}(\tau) \qquad \Gamma, x:\tau;\varphi_3 \vdash_{\mathbf{ex}} P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash_{\mathbf{ex}} \mathbf{decrypt}\ M_1\ \mathbf{is}\ \{\!|x|\!\}_{M_2{}^{-1}}.P} \quad (\text{ExT-ADec})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M_1 : \mathbf{N}_\ell(\_,\_) \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} M_2 : \mathbf{N}_\ell(\emptyset,\varphi_5) \qquad \Gamma;\varphi_3+\varphi_4+\varphi_5 \vdash_{\mathbf{ex}} P}{\Gamma;\varphi_1+\varphi_2+\varphi_3+\{\mathbf{chk}_\ell(M_1,\varphi_4)\} \vdash_{\mathbf{ex}} \mathbf{check}\ M_1\ \mathbf{is}\ M_2.P} \quad (\text{ExT-Chk})$$

$$\frac{\Gamma;\varphi+\{\mathbf{end}(V)\mapsto 1\} \vdash_{\mathbf{ex}} P}{\Gamma;\varphi \vdash_{\mathbf{ex}} \mathbf{begin}\, V.P} \quad (\text{Ext-Beg})$$

$$\frac{}{\Gamma;\varphi+\{\mathbf{end}(V)\mapsto 1\} \vdash_{\mathbf{ex}} \mathbf{end}\, V} \quad (\text{ExT-End})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M : \tau_1\times\tau_2 \qquad \Gamma, y:\tau_1, z:[y/0]\tau_2;\varphi_2 \vdash_{\mathbf{ex}} P}{\Gamma;\varphi_1+\varphi_2 \vdash_{\mathbf{ex}} \mathbf{split}\ M\ \mathbf{is}\ (y,z).P} \quad (\text{ExT-Splt})$$

$$\frac{\Gamma;\varphi_1 \vdash_{\mathbf{ex}} M_1 : \tau_1\times\tau_2 \qquad \Gamma;\varphi_2 \vdash_{\mathbf{ex}} M_2 : \tau_1 \qquad \Gamma, z:[M_2/0]\tau_2;\varphi_3 \vdash_{\mathbf{ex}} P}{\Gamma;\varphi_1+\varphi_2+\varphi_3 \vdash_{\mathbf{ex}} \mathbf{match}\ M_1\ \mathbf{is}\ (M_2,z).P} \quad (\text{ExT-Mtch})$$

$$\frac{\Gamma;\varphi' \vdash_{\mathbf{ex}} P \qquad \varphi' \le \varphi}{\Gamma;\varphi \vdash_{\mathbf{ex}} P} \quad (\text{ExT-WeakCap})$$

Figure 2.14: Typing Rules for Extended Processes

Proof. By a case analysis on the rules used for deriving $\tau_1 \leq_{\mathtt{ex}} \tau_2$ and $\tau_2 \leq_{\mathtt{ex}} \tau_3$. If one of the rules is reflexivity, the result follows immediately. There are four remaining cases.

- Case ExSubT-PubTaint-ExSubT-PubTaint: In this case $\mathbf{Pub}(\tau_1)$ and $\mathbf{Taint}(\tau_3)$, from which the result follows by ExSubT-PubTaint.
- Case ExSubT-Name-ExSubT-Name: In this case $\tau_i = \mathbf{N}_\ell(\varphi_i, \varphi_i')$ for $i \in \{1,2,3\}$ with $\varphi_1 \leq \varphi_2 \leq \varphi_3$ and $\varphi_1' \geq \varphi_2' \geq \varphi_3'$. Thus, the result follows by ExSubT-Name.
- Case ExSubT-PubTaint-ExSubT-Name: In this case we have $\tau_i = \mathbf{N}_\ell(\varphi_i, \varphi_i')$ for $i \in \{2,3\}$ with $\mathbf{Pub}(\tau_1)$, $\mathbf{Taint}(\mathbf{N}_\ell(\varphi_2, \varphi_2'))$, $\varphi_2 \leq \varphi_3$, and $\varphi_2' \geq \varphi_3$. If $\ell = \mathbf{Pub}$ then $\varphi_2' = \emptyset$, which implies $\varphi_3 = \emptyset$. Thus, we have $\ell = \mathbf{Pub} \Rightarrow \varphi_3 = \emptyset$, which implies $\mathbf{Taint}(\tau_3)$. The required result is obtained by using ExSubT-PubTaint.
- Case ExSubT-Name-ExSubT-PubTaint: In this case we have $\tau_i = \mathbf{N}_\ell(\varphi_i, \varphi_i')$ for $i \in \{1,2\}$ with $\mathbf{Taint}(\tau_3)$, $\mathbf{Pub}(\mathbf{N}_\ell(\varphi_2, \varphi_2'))$, $\varphi_1 \leq \varphi_2$, and $\varphi_1' \geq \varphi_2$. By the condition $\mathbf{Pub}(\mathbf{N}_\ell(\varphi_2, \varphi_2'))$ we have $\ell = \mathbf{Pub}$ and $\varphi_2 = \emptyset$, which implies $\varphi_1 = \emptyset$. Thus, we have $\mathbf{Pub}(\tau_1)$. The required result follows by ExSubT-PubTaint.

□

The following lemma guarantees that the subsumption rule (ExT-Sub) only increases obligations, and decreases capabilities of a name type, unless the qualification of the name type is changed from **Pub** to **Pr**.

Lemma 2.9.2. *If* $\mathbf{N}_\ell(\varphi_1, \varphi_2) \leq_{\mathtt{ex}} \mathbf{N}_{\ell'}(\varphi_1', \varphi_2')$ *then either* $\ell = \mathbf{Pub} \wedge \ell' = \mathbf{Pr} \wedge \varphi_1 = \emptyset$ *or* $\ell = \ell' \wedge \varphi_1 \leq \varphi_1' \wedge \varphi_2 \geq \varphi_2'$.

Proof. In the case where $\mathbf{N}_\ell(\varphi_1, \varphi_2) \leq_{\mathtt{ex}} \mathbf{N}_{\ell'}(\varphi_1', \varphi_2')$ was derived using rule ExSubT-Name we immediately have that the second set of conditions are satisfied. If rule ExSubT-PubTaint was used instead we first note that in this case $\ell = \mathbf{Pub}$ must hold. Then, if $\ell' = \mathbf{Pub}$ (respectively **Pr**) we have that the second (respectively first) set of conditions are satisfied. □

Extended Operational Semantics

The set of *message reduction contexts* for messages, ranged over by C_m, is given by:

$$C_m ::= [\,] \mid \mathbf{addC}(C_m, \varphi_1, \varphi_2) \mid (C_m, M) \mid (V, C_m) \mid \{C_m\}_M \mid \{V\}_{C_m} \mid \{\!|C_m|\!\}_M \mid \{\!|V|\!\}_{C_m}$$

and the message reduction relation by rules:

$$(\varphi + \varphi_1' + \varphi_2', C_m[\mathbf{addC}(n^{(\varphi_1,\varphi_2)}, \varphi_1', \varphi_2')]) \longrightarrow_{\mathbf{ex}} (\varphi, C_m[n^{(\varphi_1 - \varphi_1', \varphi_2 + \varphi_2')}])$$

$$(\varphi, C_m[\mathbf{addC}(n^{(\varphi_1,\varphi_2)}, \varphi_1', \varphi_2')]) \longrightarrow_{\mathbf{ex}} \mathbf{Error} \quad \text{if } \varphi_1' + \varphi_2' \not\leq \varphi$$

The extended reduction relation $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle \longrightarrow_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi' \rangle$ is defined by the rules in Figure 2.15. Here, Ψ is a multiset of extended processes, E is a set of messages (that represent the begin-events that have occurred but have not been matched by corresponding end-events), Γ is a set of names that have been created (with type assumptions), $\mathcal{K}$ is a set of pairs of decryption and encryption keys, and φ is a capability. In the figure, C ranges over the

$$\frac{(\varphi, M) \longrightarrow_{\mathbf{ex}} (\varphi', M')}{\langle \Psi \uplus \{C[M]\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{C[M']\}, E, \Gamma, \mathcal{K}, \varphi'\rangle} \quad \text{(ExR-M)}$$

$$\frac{(\varphi, M) \longrightarrow_{\mathbf{ex}} \mathbf{Error}}{\langle \Psi \uplus \{C[M]\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \mathbf{Error}} \quad \text{(ExR-M-Er)}$$

$$\langle \Psi \uplus \{n^{(\_,\_)}?y.P, n^{(\_,\_)}!V\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[V/y]P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-Com)}$$

$$\langle \Psi \uplus \{P \mid Q\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{P, Q\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-Par)}$$

$$\langle \Psi \uplus \{*P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{*P, P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-Rep)}$$

$$\langle \Psi \uplus \{(\nu x : \mathbf{N}_\ell(\varphi_1, \emptyset))P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[n^{(\varphi_1,\emptyset)}/x]P\}, E, \Gamma \cup \{n : \mathbf{N}_\ell(\varphi_1, \emptyset)\}, \mathcal{K}, \varphi + \{\mathbf{chk}_\ell(n, \varphi_1)\}\rangle \quad (n \notin dom(\Gamma)) \quad \text{(ExR-NewN)}$$

$$\langle \Psi \uplus \{(\nu_{sym} x : \tau)P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[k/x]P\}, E, \Gamma \cup \{k : \tau\}, \mathcal{K}, \varphi\rangle \ (k \notin dom(\Gamma)) \quad \text{(ExR-NewSk)}$$

$$\langle \Psi \uplus \{(\nu_{asym} x : \tau_1, y : \tau_2)P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[k_1/x, k_2/y]P\}, E, \Gamma \cup \{k_1 : \tau_1, k_2 : \tau_2\}, \mathcal{K} \cup \{(k_1, k_2)\}, \varphi\rangle \ (k_1, k_2 \notin dom(\Gamma)) \quad \text{(ExR-NewAk)}$$

$$\langle \Psi \uplus \{\mathbf{check}\ n^{(\_,\_)}\ \mathbf{is}\ n^{(\emptyset,\varphi_1)}.P\}, E, \Gamma, \mathcal{K}, \varphi + \{\mathbf{chk}_\ell(n, \varphi_2)\}\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{P\}, E, \Gamma, \mathcal{K}, \varphi + \varphi_1 + \varphi_2\rangle \quad \text{(ExR-Chk)}$$

$$\frac{(\varphi_0 \neq \emptyset) \vee \neg\exists \varphi_2, \ell.(\mathbf{chk}_\ell(n, \varphi_2) \in \varphi)}{\langle \Psi \uplus \{\mathbf{check}\ n^{(\_,\_)}\ \mathbf{is}\ n^{(\varphi_0,\varphi_1)}.P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \mathbf{Error}} \quad \text{(ExR-Chk-Er)}$$

$$\langle \Psi \uplus \{\mathbf{split}\ (V, W)\ \mathbf{is}\ (x, y).P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[V/x, W/y]P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-Splt)}$$

$$\langle \Psi \uplus \{\mathbf{match}\ (V, W)\ \mathbf{is}\ (V, z).P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[W/z]P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-Mtch)}$$

$$\langle \Psi \uplus \{\mathbf{decrypt}\ \{V\}_k\ \mathbf{is}\ \{x\}_k.P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[V/x]P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-SDec)}$$

$$\frac{(k_1, k_2) \in \mathcal{K}}{\langle \Psi \uplus \{\mathbf{decrypt}\ \{\!|V|\!\}_{k_1}\ \mathbf{is}\ \{\!|x|\!\}_{k_2{}^{-1}}.P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{[V/x]P\}, E, \Gamma, \mathcal{K}, \varphi\rangle} \quad \text{(ExR-ADec)}$$

$$\langle \Psi \uplus \{\mathbf{begin}\, V.P\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi \uplus \{P\}, E \uplus \{V\}, \Gamma, \mathcal{K}, \varphi + \{\mathbf{end}(V)\}\rangle \quad \text{(ExR-Beg)}$$

$$\langle \Psi \uplus \{\mathbf{end}\, V\}, E \uplus \{V\}, \Gamma, \mathcal{K}, \varphi + \{\mathbf{end}(V)\}\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle \quad \text{(ExR-End)}$$

$$\frac{(V \notin E) \vee (\varphi(\mathbf{end}(V)) < 1)}{\langle \Psi \uplus \{\mathbf{end}\, V\}, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \mathbf{Error}} \quad \text{(ExR-End-Er)}$$

Figure 2.15: Extended Operational Semantics

set of process reduction contexts given by:

$$\begin{aligned} C ::= {} & M_1!M_2 \mid v!C_m \mid M?x.P \mid (P_1 \mid P_2) \\ & \mid \mathbf{check}\ C_m\ \mathbf{is}\ M.P \mid \mathbf{check}\ V\ \mathbf{is}\ C_m.P \\ & \mid \mathbf{split}\ C_m\ \mathbf{is}\ (x, y).P \\ & \mid \mathbf{match}\ C_m\ \mathbf{is}\ (M, y).P \mid \mathbf{match}\ V\ \mathbf{is}\ (C_m, y).P \\ & \mid \mathbf{decrypt}\ C_m\ \mathbf{is}\ \{x\}_{M_2}.P \mid \mathbf{decrypt}\ V\ \mathbf{is}\ \{x\}_{C_m}.P \\ & \mid \mathbf{decrypt}\ C_m\ \mathbf{is}\ \{\!|x|\!\}_{M_2{}^{-1}}.P \\ & \mid \mathbf{decrypt}\ V\ \mathbf{is}\ \{\!|x|\!\}_{C_m{}^{-1}}.P \end{aligned}$$

Proof of Lemma

For an extended process P we write $\mathbf{Erase}(P)$ for the process obtained by removing type annotations and "addC".

Lemma 2.9.3. *If* $\Gamma;\varphi \vdash P$ *then there exists* P' *such that* $\Gamma;\varphi \vdash_{\mathbf{ex}} P'$ *and* $\mathbf{Erase}(P') = P$.

Proof. Easy induction on the derivation of $\Gamma;\varphi \vdash P$. □

Lemma 2.9.4. *If* $\langle P,\emptyset,\emptyset,\emptyset,\emptyset\rangle \not\longrightarrow^*_{\mathbf{ex}} \mathbf{Error}$ *then* $\mathbf{Erase}(P)$ *is safe.*

Proof. We show contraposition. Suppose $\mathbf{Erase}(P)$ is not safe, i.e. $\langle \mathbf{Erase}(P),\emptyset,\emptyset,\emptyset\rangle \longrightarrow^* \mathbf{Error}$. Then $\langle P,\emptyset,\emptyset,\emptyset,\emptyset\rangle \longrightarrow^*_{\mathbf{ex}} \mathbf{Error}$ follows from the facts: (i) if $\langle \mathbf{Erase}(P), E, dom(\Gamma), \mathcal{K}\rangle \longrightarrow \langle Q, E', N', \mathcal{K}'\rangle$, then either $\langle P, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle P', E', \Gamma', \mathcal{K}', \varphi'\rangle$ with $\mathbf{Erase}(P') = P$ and $dom(\Gamma') = N'$ or $\langle P, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \mathbf{Error}$; and (ii) if $\langle \mathbf{Erase}(P), E, dom(\Gamma), \mathcal{K}\rangle \longrightarrow \mathbf{Error}$, then $\langle P, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow^*_{\mathbf{ex}} \mathbf{Error}$. (These facts follow by an easy case analysis on the rule used for deriving $\langle \mathbf{Erase}(P), E, dom(\Gamma), \mathcal{K}\rangle \longrightarrow \langle Q, E', N', \mathcal{K}'\rangle$ or $\langle \mathbf{Erase}(P), E, dom(\Gamma), \mathcal{K}\rangle \longrightarrow \mathbf{Error}$.) □

Lemma 2.9.5. *If* $\emptyset;\emptyset \vdash_{\mathbf{ex}} P$ *then* $\langle P,\emptyset,\emptyset,\emptyset,\emptyset\rangle \not\longrightarrow^*_{\mathbf{ex}} \mathbf{Error}$.

To show Lemma 2.9.5 we define a typing rule for run-time configurations (of the form $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$) and show (i) the typing is preserved (Lemma 2.9.9) and (ii) a well-typed configuration is not immediately reduced to **Error** (Lemma 2.9.6).

In order to express a necessary invariant we introduce a reduction relation $(\varphi, N) \Longrightarrow_\Psi (\varphi', N')$ used to collect all the capabilities, including those attached to names, and defined by:

$$\frac{n^{(\varphi_3,\varphi_4)} \text{ occurs in } \Psi}{(\varphi_1 + \{\mathbf{chk}_\ell(n,\varphi_2) \mapsto 1\}, N) \Longrightarrow_\Psi (\varphi_1 + (\varphi_2 - \varphi_3) + \varphi_4, N \cup \{n\})}$$

where the second component N is a set of names keeping track of the names already checked.

We write $\mathbf{ConsistentCap}(E,\varphi,\Psi)$ if, whenever $(\varphi,\emptyset) \Longrightarrow^*_\Psi (\varphi', N)$, the following conditions hold: (i) $E(V) \geq \varphi'(\mathbf{end}(V))$ for every V, and (ii) for every n, if $\varphi'(\mathbf{chk}_\ell(n,\varphi_2)) \geq 1$, then $n \notin N$. The condition (i) ensures that the end capabilities estimated by the type system (i.e. φ') is at most those that are actually present (E). The condition (ii) ensures that there is always at most one check capability for each name.

Let the typing rule for run-time configurations be given by:

$$\frac{\begin{array}{c}\Gamma;\varphi \vdash_{\mathbf{ex}} P_1 \mid \cdots \mid P_m \\ \mathbf{ConsistentCap}(E,\varphi,\{P_1,\ldots,P_m\}) \\ \forall (k_1,k_2) \in \mathcal{K}.\exists \tau.(\Gamma(k_1) = \mathbf{EKey}(\tau) \wedge \Gamma(k_2) = \mathbf{DKey}(\tau)) \\ \forall n,\ell,\ell'.(\Gamma(n) = \mathbf{N}_\ell(\_,\_) \wedge \\ (\text{“}\mathbf{chk}_{\ell'}(n,\_) \text{ occurs in some } P_i \text{ or } \varphi\text{”}) \Rightarrow \ell = \ell')\end{array}}{\vdash_{\mathbf{ex}} \langle \{P_1,\ldots,P_m\}, E, \Gamma, \mathcal{K}, \varphi\rangle}$$

Lemma 2.9.6 (lack of immediate error)**.** *If* $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ *then* $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle \not\longrightarrow_{\mathbf{ex}} \mathbf{Error}$.

Proof. Suppose $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ holds. There are three rules that may yield **Error**: ExR-M-Er, ExR-Chk-Er, and ExR-End-Er. We show below that none of those rules are applicable.

- Case ExR-M-Er: In this case $\Psi = \Psi' \uplus \{C[M]\}$ with $(\varphi, M) \longrightarrow_{\mathbf{ex}} \mathbf{Error}$. By the typing rules it must be the case that $\Gamma;\varphi' \vdash_{\mathbf{ex}} M : \tau$ and $\varphi' \subseteq \varphi$ for some φ' and τ. By the typing rules and reduction rules for messages, $(\varphi, M) \longrightarrow_{\mathbf{ex}} \mathbf{Error}$ cannot hold.

- Case ExR-Chk-Er: In this case $\Psi = \Psi' \uplus \{\mathbf{check}\ n^{(\_,\_)}\ \mathbf{is}\ n^{(\varphi_0,\varphi_1)}.P\}$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ there must exist φ' such that $\varphi' \leq \varphi$ and $\Gamma; \varphi' \vdash_{\mathbf{ex}} \mathbf{check}\ n^{(\_,\_)}\ \mathbf{is}\ n^{(\varphi_0,\varphi_1)}.P$. By the typing rules we have:

$$\begin{array}{l}\Gamma; \varphi_2 \vdash_{\mathbf{ex}} n^{(\_,\_)} : \mathbf{N}_\ell(\_,\_)\\ \Gamma; \varphi_3 \vdash_{\mathbf{ex}} n^{(\varphi_0,\varphi_1)} : \mathbf{N}_\ell(\emptyset, \varphi_5)\\ \Gamma; \varphi_4 + \varphi_5 + \varphi_6 \vdash_{\mathbf{ex}} P\\ \varphi' \geq \varphi_2 + \varphi_3 + \varphi_4 + \{\mathbf{chk}_\ell(n, \varphi_6)\}\end{array}$$

  and by the second condition that $\varphi_0 = \emptyset$. (Note that the judgment must have been derived from ExT-Name, followed by a possible application of ExT-Sub. ExT-Name assigns the type $\mathbf{N}_{\ell'}(\varphi_0, \varphi_1)$, and by Lemma 2.9.1 we must have $\mathbf{N}_{\ell'}(\varphi_0, \varphi_1) \leq_{\mathbf{ex}} \mathbf{N}_\ell(\emptyset, \varphi_5)$. By Lemma 2.9.2 we have $\varphi_0 = \emptyset$. Thus, the premise of ExR-Chk-Er does not hold.

- Case ExR-End-Er: In this case $\Psi = \Psi' \uplus \{\mathbf{end}\, V\}$ with $(V \notin E) \vee (\varphi(\mathbf{end}(V)) < 1)$. If $V \notin E$ then by the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ and the second condition on the configuration typing we have $(\varphi(\mathbf{end}(V)) < 1)$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$, however, we also have $\Gamma; \varphi' \vdash_{\mathbf{ex}} \mathbf{end}\, V$ for some $\varphi' \leq \varphi$. By the typing rule for $\mathbf{end}\, V$ it must be the case that $(\varphi'(\mathbf{end}(V)) \geq 1)$, hence a contradiction.

□

Lemma 2.9.7. *If* $\Gamma; \varphi \vdash_{\mathbf{ex}} V : \tau$ *then* $\Gamma; \emptyset \vdash_{\mathbf{ex}} V : \tau$.

Proof. Straightforward induction on the derivation of $\Gamma; \varphi \vdash_{\mathbf{ex}} V : \tau$. (Note that by the syntax of values, V does not contain "addC".) □

Lemma 2.9.8 (substitution)**.** *If* $\Gamma_1; \emptyset \vdash_{\mathbf{ex}} V : \tau$ *and* $\Gamma_1, x{:}\tau, \Gamma_2; \varphi \vdash_{\mathbf{ex}} P$ *then* $\Gamma_1, [V/x]\Gamma_2; [V/x]\varphi \vdash_{\mathbf{ex}} [V/x]P$.

Proof. A derivation of $\Gamma_1, [V/x]\Gamma_2; [V/x]\varphi \vdash_{\mathbf{ex}} [V/x]P$ is obtained from $\Gamma_1, x:\tau, \Gamma_2; \varphi \vdash_{\mathbf{ex}} P$ by replacing each leaf of the form $\Gamma_1, x{:}\tau, \Gamma_2'; \varphi' \vdash_{\mathbf{ex}} x{:}\tau$ (where $\Gamma_2' \supseteq \Gamma_2$) with $\Gamma_1, [V/x]\Gamma_2; [V/x]\varphi' \vdash_{\mathbf{ex}} V : \tau$ (which is obtained by weakening and ExT-WeakCap). □

Lemma 2.9.9 (type preservation)**.** *If* $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ *and* $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi'\rangle$ *then* $\vdash_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi'\rangle$.

Proof. Suppose $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$ and $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi'\rangle$. We show $\vdash_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi'\rangle$ by case analysis on the rule used for deriving $\langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle \longrightarrow_{\mathbf{ex}} \langle \Psi', E', \Gamma', \mathcal{K}', \varphi'\rangle$. By abuse of notation we often write $\Gamma; \varphi \vdash \{P_1, \ldots, P_k\}$ for $\Gamma; \varphi \vdash P_1 \mid \cdots \mid P_k$.

- Case ExR-M: In this case $\Psi = \Psi_1 \uplus \{C[M]\}$ and $\Psi' = \Psi_1 \uplus \{C[M']\}$ with $M = C_m[\mathbf{addC}(n^{(\varphi_1,\varphi_2)}, \varphi_1', \varphi_2')]$, $M' = C_m[n^{(\varphi_1-\varphi_1', \varphi_2+\varphi_2')}]$, and $\varphi = \varphi' + \varphi_1' + \varphi_2'$. We also have $E' = E$, $\Gamma' = \Gamma$, and $\mathcal{K}' = \mathcal{K}$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi\rangle$, $\Gamma; \varphi \vdash \Psi_1 \uplus \{C[C_m[\mathbf{addC}(n^{(\varphi_1,\varphi_2)}, \varphi_1', \varphi_2')]]\}$ holds, which must have been derived from $\Gamma; \varphi_1' + \varphi_2' \vdash_{\mathbf{ex}} \mathbf{addC}(n^{(\varphi_1,\varphi_2)}, \varphi_1', \varphi_2') : \mathbf{N}_{\ell'}(\varphi_1'' - \varphi_1', \varphi_2'' + \varphi_2')$, where $\Gamma(n) = \mathbf{N}_\ell(\_,\_)$ and $\mathbf{N}_\ell(\varphi_1, \varphi_2) \leq_{\mathbf{ex}} \mathbf{N}_{\ell'}(\varphi_1'', \varphi_2'')$. By Lemma 2.9.2 we have either $\ell = \mathbf{Pub} \wedge \ell' = \mathbf{Taint} \wedge \varphi_1 = \emptyset$ or $\ell = \ell' \wedge \varphi_1 \leq \varphi_1'' \wedge \varphi_2 \geq \varphi_2''$. In both cases we have $\mathbf{N}_\ell(\varphi_1 - \varphi_1', \varphi_2 + \varphi_2') \leq_{\mathbf{ex}} \mathbf{N}_{\ell'}(\varphi_1'' - \varphi_1', \varphi_2'' + \varphi_2')$, which implies $\Gamma; \emptyset \vdash_{\mathbf{ex}} n^{(\varphi_1-\varphi_1', \varphi_2+\varphi_2')} : \mathbf{N}_{\ell'}(\varphi_1'' - \varphi_1', \varphi_2'' + \varphi_2')$. Thus, we have

$$\Gamma; \varphi' \vdash \Psi_1 \uplus \{C[C_m[n^{(\varphi_1-\varphi_1', \varphi_2+\varphi_2')}]]\}.$$

  It remains to check $\mathbf{ConsistentCap}(E, \varphi', \Psi')$. To check this it suffices to observe that whenever $(\varphi', \emptyset) \Longrightarrow_{\Psi'}^* (\varphi_3', \varphi_4')$, we can construct a corresponding sequence $(\varphi, \emptyset) \Longrightarrow_{\Psi}^* (\varphi_3, \varphi_4)$ such that $\varphi_3' + \varphi_4' \leq \varphi_3 + \varphi_4$. (The only reduction step $(\varphi', \emptyset) \Longrightarrow_{\Psi'}^* (\varphi_3', \varphi_4')$

introduces more capabilities is a reduction on $\mathbf{chk}_\ell(n, \_)$, but that can happen at most once, and the difference is at most $\varphi_1' + \varphi_2'$.)

- Case ExR-Com: In this case $\Psi = \Psi_1 \uplus \{n^{(\_,\_)}?y.P, n^{(\_,\_)}!V\}$ and $\Psi = \Psi_1 \uplus \{[V/y]P\}$, with $E' = E$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$, and $\varphi' = \varphi$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l}\Gamma; \varphi_1 \vdash_{\mathbf{ex}} \Psi_1 \\ \Gamma; \varphi_2 \vdash_{\mathbf{ex}} n^{(\_,\_)} : \mathbf{N}_\ell(\emptyset, \emptyset) \\ \Gamma; \varphi_3 \vdash_{\mathbf{ex}} V : \tau \\ \mathbf{Pub}(\tau) \\ \Gamma; \varphi_4 \vdash_{\mathbf{ex}} n^{(\_,\_)} : \mathbf{N}_{\ell'}(\emptyset, \emptyset) \\ \Gamma, y : \tau'; \varphi_5 \vdash_{\mathbf{ex}} P \\ \mathbf{Taint}(\tau') \\ \varphi \geq \varphi_1 + \varphi_2 + \varphi_3 + \varphi_4 + \varphi_5\end{array}$$

  By the conditions $\Gamma; \varphi_3 \vdash_{\mathbf{ex}} V : \tau$, $\mathbf{Pub}(\tau)$, and $\mathbf{Taint}(\tau')$, we have $\Gamma; \varphi_3 \vdash_{\mathbf{ex}} V : \tau'$. By Lemma 2.9.7 $\Gamma; \emptyset \vdash_{\mathbf{ex}} V : \tau'$ holds. Thus, by using the substitution lemma (Lemma 2.9.8) we obtain $\Gamma; \varphi_5 \vdash_{\mathbf{ex}} [V/y]P$. By using ExT-Par and ExT-WeakCap we obtain $\Gamma; \varphi \vdash_{\mathbf{ex}} \Psi'$ as required. $\mathbf{ConsistentCap}(E, \varphi, \Psi')$ follows immediately from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-Par, ExR-Rep: Trivial.

- Case ExR-NewN: In this case $\Psi = \Psi_1 \uplus \{(\nu x{:}\mathbf{N}_\ell(\varphi_1, \emptyset))P\}$ and $\Psi' = \Psi_1 \uplus \{[n^{(\varphi_1,\emptyset)}/x]P\}$, with $E' = E$, $\Gamma' = (\Gamma, n : \mathbf{N}_\ell(\varphi_1, \emptyset))$, $\mathcal{K}' = \mathcal{K}$, and $\varphi' = \varphi + \{\mathbf{chk}_\ell(n, \varphi_1) \mapsto 1\}$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ and rule ExT-NewN we have: $\Gamma; \varphi_2 \vdash_{\mathbf{ex}} \Psi_1$ and $\Gamma, x : \mathbf{N}_\ell(\varphi_1, \emptyset); \varphi_3 + \{\mathbf{chk}_\ell(x, \varphi_1) \mapsto 1\} \vdash_{\mathbf{ex}} P$, with $\varphi \geq \varphi_2 + \varphi_3$. By the substitution lemma (Lemma 2.9.8) we have

$$\Gamma'; \varphi_3 + \{\mathbf{chk}_\ell(n, \varphi_1) \mapsto 1\} \vdash_{\mathbf{ex}} [n^{(\varphi_1,\emptyset)}/x]P.$$

  Thus, by using ExT-Par and ExT-WeakCap we obtain $\Gamma'; \varphi' \vdash_{\mathbf{ex}} \Psi'$ as required. $\mathbf{ConsistentCap}(E', \varphi', \Psi')$ follows immediately from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-NewSk: Similar to the case for ExR-New.

- Case ExR-NewAk: Similar to the case for ExR-New.

- Case ExR-Chk: In this case $\Psi = \Psi_1 \uplus \{\mathbf{check}\ n^{(\_,\_)}\ \mathbf{is}\ n^{(\emptyset,\varphi_1)}.P\}$ and $\Psi' = \Psi_1 \uplus \{P\}$, with $E' = E$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$, $\varphi = \varphi_0 + \{\mathbf{chk}_\ell(n, \varphi_2)\}$, and $\varphi' = \varphi_0 + \varphi_1 + \varphi_2$.

  By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l}\Gamma; \varphi_3 \vdash_{\mathbf{ex}} \Psi_1 \\ \Gamma; \varphi_4 \vdash_{\mathbf{ex}} n^{(\_,\_)} : \mathbf{N}_\ell(\_, \_) \\ \Gamma; \varphi_5 \vdash_{\mathbf{ex}} n^{(\varphi_7,\varphi_8)} : \mathbf{N}_\ell(\emptyset, \varphi_1) \\ \qquad (\text{with } \mathbf{N}_\ell(\varphi_7, \varphi_8) \leq_{\mathbf{ex}} \mathbf{N}_\ell(\emptyset, \varphi_1)) \\ \Gamma; \varphi_6 + \varphi_1 + \varphi_2 \vdash_{\mathbf{ex}} P \\ \varphi_0 \geq \varphi_3 + \varphi_4 + \varphi_5 + \varphi_6\end{array}$$

  Therefore, we have $\Gamma; \varphi_3 + \varphi_6 + \varphi_1 + \varphi_2 \vdash_{\mathbf{ex}} \Psi'$. By ExT-WeakCap we obtain $\Gamma; \varphi' \vdash_{\mathbf{ex}} \Psi'$. It remains to check $\mathbf{ConsistentCap}(E, \varphi', \Psi')$.

  Next, we show that $\varphi_7 = \emptyset$ and $\varphi_8 \geq \varphi_1$. By the condition $\mathbf{N}_\ell(\varphi_7, \varphi_8) \leq_{\mathbf{ex}} \mathbf{N}_\ell(\emptyset, \varphi_1)$ either $(\varphi_7 = \emptyset) \wedge (\varphi_8 = \varphi_1)$ or $\mathbf{Pub}(\mathbf{N}_\ell(\varphi_7, \varphi_8)) \wedge \mathbf{Taint}(\mathbf{N}_\ell(\emptyset, \varphi_1))$ holds. In the latter case $\ell = \mathbf{Pub}$ and $\varphi_7 = \varphi_1 = \emptyset$. Thus, we have $\varphi_7 = \emptyset$ and $\varphi_8 \geq \varphi_1$ as required.

  Since $(\varphi, \emptyset) \Longrightarrow_\Psi (\varphi_0 + \varphi_2 + \varphi_1', \{\mathbf{chk}_\ell(n, \varphi_2)\})$ for some $\varphi_1' \geq \varphi_8 \geq \varphi_1$, $\mathbf{ConsistentCap}(E, \varphi', \Psi')$ follows from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-Splt: In this case $\Psi = \Psi_1 \uplus \{\mathbf{split}\ (V, W)\ \mathbf{is}\ (x, y).P\}$ and $\Psi' = \Psi_1 \uplus \{[V/x, W/y]P\}$, with $E' = E$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$, and $\varphi' = \varphi$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l} \Gamma; \varphi_1 \vdash_{\mathbf{ex}} \Psi_1 \\ \Gamma; \varphi_2 \vdash_{\mathbf{ex}} V : \tau_1 \\ \Gamma; \varphi_3 \vdash_{\mathbf{ex}} W : [V/0]\tau_2 \\ \Gamma, x : \tau_1, y : [x/0]\tau_2; \varphi_4 \vdash_{\mathbf{ex}} P \\ \varphi \geq \varphi_1 + \varphi_2 + \varphi_3 + \varphi_4 \end{array}$$

  Here, φ_4 does not contain x and y. Without loss of generality we also assume that x, y does not occur in V, W. By applying the substitution lemma (Lemma 2.9.8) we have $\Gamma, y : [V/0]\tau_2; \varphi_4 \vdash_{\mathbf{ex}} [V/x]P$. By applying the substitution lemma again we get: $\Gamma; \varphi_4 \vdash_{\mathbf{ex}} [V/x, W/y]P$. Thus, we obtain $\Gamma'; \varphi' \vdash_{\mathbf{ex}} \Psi'$ as required. $\mathbf{ConsistentCap}(E, \varphi', \Psi')$ follows from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-Mtch: Similar to the case ExR-Splt above.

- Case ExR-SDec: Similar to the case ExR-ADec below.

- Case ExR-ADec: In this case $\Psi = \Psi_1 \uplus \{\mathbf{decrypt}\ \{\!|V|\!\}_{k_1}\ \mathbf{is}\ \{\!|x|\!\}_{k_2{}^{-1}}.P\}$ and $\Psi' = \Psi_1 \uplus \{[V/x]P\}$, with $E' = E$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$, and $\varphi' = \varphi$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l} \Gamma; \varphi_1 \vdash_{\mathbf{ex}} \Psi_1 \\ \Gamma; \varphi_2 \vdash_{\mathbf{ex}} k_1 : \mathbf{EKey}(\tau_1) \\ \Gamma; \varphi_3 \vdash_{\mathbf{ex}} V : \tau_1 \\ \Gamma; \varphi_4 \vdash_{\mathbf{ex}} k_2 : \mathbf{DKey}(\tau_2) \\ \Gamma, x : \tau_2; \varphi_5 \vdash_{\mathbf{ex}} P \\ \varphi \geq \varphi_1 + \varphi_2 + \varphi_3 + \varphi_4 + \varphi_5 \\ \Gamma(k_1) = \mathbf{EKey}(\tau) \\ \Gamma(k_2) = \mathbf{DKey}(\tau) \end{array}$$

  By the 2nd, 4th, and the last two conditions, we have $\mathbf{EKey}(\tau) \leq_{\mathtt{ex}} \mathbf{EKey}(\tau_1)$ and $\mathbf{DKey}(\tau) \leq_{\mathtt{ex}} \mathbf{DKey}(\tau_2)$. $\mathbf{EKey}(\tau) \leq_{\mathtt{ex}} \mathbf{EKey}(\tau_1)$ implies $\tau = \tau_1$ or $\mathbf{Pub}(\mathbf{EKey}(\tau)) \wedge \mathbf{Taint}(\mathbf{EKey}(\tau_1))$, which implies $\tau = \tau_1$ or $\mathbf{Taint}(\tau) \wedge \mathbf{Pub}(\tau_1)$. Thus, we have $\tau_1 \leq_{\mathtt{ex}} \tau$. Similarly, $\mathbf{DKey}(\tau) \leq_{\mathtt{ex}} \mathbf{DKey}(\tau_2)$ implies $\tau \leq_{\mathtt{ex}} \tau_2$. As the subtyping relation is transitive (Lemma 2.9.1) we have $\tau_1 \leq_{\mathtt{ex}} \tau_2$. Thus, by using ExT-Sub and the substitution lemma (Lemma 2.9.8) we obtain $\Gamma; \varphi_5 \vdash_{\mathbf{ex}} [V/x]P$. By Ext-WeakCap we obtain $\Gamma; \varphi \vdash_{\mathbf{ex}} \Psi'$ as required.

  $\mathbf{ConsistentCap}(E', \varphi', \Psi')$ follows immediately from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-Beg: In this case $\Psi = \Psi_1 \uplus \{\mathbf{begin}\, V.P\}$ and $\Psi' = \Psi_1 \uplus \{P\}$, with $E' = E \uplus \{\mathbf{end}(V)\}$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$, and $\varphi' = \varphi + \{\mathbf{end}(V) \mapsto 1\}$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l} \Gamma; \varphi_1 \vdash_{\mathbf{ex}} \Psi_1 \\ \Gamma; \varphi_2 + \{\mathbf{end}(V) \mapsto 1\} \vdash_{\mathbf{ex}} P \\ \varphi \geq \varphi_1 + \varphi_2 \end{array}$$

  Thus, we have $\Gamma; \varphi' \vdash_{\mathbf{ex}} \Psi'$ as required. $\mathbf{ConsistentCap}(E', \varphi', \Psi')$ follows immediately from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

- Case ExR-End: In this case $\Psi = \Psi' \uplus \{\mathbf{end}\, V\}$ with $E = E' \uplus \{\mathbf{end}(V)\}$, $\Gamma' = \Gamma$, $\mathcal{K}' = \mathcal{K}$,

and $\varphi = \varphi' + \{\mathbf{end}(V) \mapsto 1\}$. By the assumption $\vdash_{\mathbf{ex}} \langle \Psi, E, \Gamma, \mathcal{K}, \varphi \rangle$ we have:

$$\begin{array}{l} \Gamma; \varphi_1 \vdash_{\mathbf{ex}} \Psi' \\ \Gamma; \varphi_2 + \{\mathbf{end}(V) \mapsto 1\} \vdash_{\mathbf{ex}} \mathbf{end}\, V \\ \varphi \geq \varphi_1 + \varphi_2 + \{\mathbf{end}(V) \mapsto 1\} \end{array}$$

Thus, we have $\Gamma; \varphi \vdash_{\mathbf{ex}} \Psi'$ as required. $\mathbf{ConsistentCap}(E', \varphi', \Psi')$ follows immediately from $\mathbf{ConsistentCap}(E, \varphi, \Psi)$.

□

Lemma 2.3.2 now follows as an immediate corollary of the lemmas above.

Proof of Lemma 2.3.2 Suppose $\emptyset; \emptyset \vdash P$. By Lemma 2.9.3 there exists an extended process P' such that $\emptyset; \emptyset \vdash_{\mathbf{ex}} P'$ and $\mathbf{Erase}(P') = P$. By Lemma 2.9.5 $\langle P', \emptyset, \emptyset, \emptyset, \emptyset \rangle \not\rightarrow^*_{\mathbf{ex}} \mathbf{Error}$. Thus, by Lemma 2.9.4 and $P = \mathbf{Erase}(P')$ we get that P is safe. □

2.10 Bibliography

[Aba99] Martín Abadi. Secrecy by typing in security protocols. *JACM*, 46(5):749–786, 1999.

[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.

[BBF+08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 17–32, 2008.

[BFG10] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *Proceedings of POPL 2010*, pages 445–456, 2010.

[BFM05] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Analysis of typed analyses of authentication protocols. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*, pages 112–125, 2005.

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[Bla02] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer-Verlag, 2002.

[CM06] Cas J. F. Cremers and Sjouke Mauw. A family of multi-party authentication protocols - extended abstract. In *Proceedings of WISSEC'06*, 2006.

[Cre08] Cas J. F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *Proceedings of ACM Conference on Computer and Communications Security (CCS 2008)*, pages 119–128, 2008.

[FGM07] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Prog. Lang. Syst.*, 29(5), 2007.

[FMP05] Riccardo Focardi, Matteo Maffei, and Francesco Placella. Inferring authentication tags. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS 2005)*, pages 41–49, 2005.

[GJ02a] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. Technical Report MRS-TR-2002-31, Microsoft Research, August 2002.

[GJ02b] Andrew D. Gordon and Alan Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium (ISSS 2002)*, volume 2609 of *LNCS*, pages 263–282. Springer-Verlag, 2002.

[GJ03] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

[GJ04] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

[HJ04] Christian Haack and Alan Jeffrey. Cryptyc. http://www.cryptyc.org/, 2004.

[KK07] Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Proceedings of APLAS 2007*, volume 4807 of *LNCS*, pages 191–205. Springer-Verlag, 2007.

[KK09] Daisuke Kikuchi and Naoki Kobayashi. Type-based automated verification of authenticity in cryptographic protocols. In *Proceedings of ESOP 2009*, volume 5502 of *LNCS*, pages 222–236. Springer-Verlag, 2009.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[WL93] Thomas Y.C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–193, 1993.

Chapter 3

Privacy for Anonymous Location Based Services

We propose a framework for formal analysis of privacy in location based services such as anonymous electronic toll collection. We give a formal definition of privacy, and apply it to the VPriv scheme for vehicular services. We analyse the resulting model using the ProVerif tool, concluding that our privacy property holds only if certain conditions are met by the implementation. Our analysis includes some novel features such as the formal modelling of privacy for a protocol that relies on interactive zero-knowledge proofs of knowledge and list permutations.

3.1 Introduction

The sophistication and quantity of embedded devices in modern vehicles is growing rapidly. Ad-hoc wireless networking is envisioned as one of the next big steps, with various car-to-infrastructure and car-to-car communication applications planned [Sta06, DINI05]. Many of these applications are location-based, and providing the precise position of the vehicle is essential to the quality of the service provided. As these applications are deployed, privacy concerns naturally emerge.

Some of the location-based services already in widespread use today, such as RFID tag based electronic toll collection systems, offer little privacy protection to drivers [Law08]. By using the same fixed identifying tag whenever they have to pay a toll fee, it becomes trivial to later trace the routes of any driver given the database of payments. Little is gained by using a fixed random tag instead of a real-world identifier such as the license plate. Although the tolling database may not be publicly available, the privacy of drivers is still at risk of exploitation from within the toll company.

The more widespread employment of such systems combined with the possibility of moving them to the emerging general framework for network communication, increases the need for privacy oriented systems. In this chapter, we bring the privacy analysis of location-based services into the world of formal methods, leveraging previous work on privacy for vehicular mix-zones [DDS10], electronic voting [KR05, DRS08], and RFID tags [ACRR10, BCdH10]. In particular, we concentrate on VPriv [BBP09], a proposed scheme for building location-based services using zero-knowledge techniques, designed to ensure that the paths of drivers are not revealed to the service providers, while nonetheless preventing drivers from reporting fake paths. We use the formal notion of indistinguishability to formalise privacy and carry out the analysis with the aid of the protocol analysis tool ProVerif [Bla04]. In particular, we will use a notion of trace equivalence, after explaining why the more usual notion of observational equivalence is not suitable in this setting. To the best of our knowledge this is the first use of the tool for analysing

a protocol that relies on interactive zero-knowledge proofs of knowledge. Note that contrary to non-interactive ones that can be abstracted by means of an appropriate equational theory (see *e.g.* [BMU08]), interactive proofs cannot since the interactions between the participants reveal some information that has to be considered when we carry out the privacy analysis.

3.2 The VPriv Scheme

In this section we introduce the VPriv scheme [BBP09], a protocol that offers a variety of location-based vehicular services such as "pay-as-you-go" insurance, electronic toll collection, etc. Its goal is to both protect the privacy of drivers whilst ensuring that they cannot cheat service providers by, for instance, paying a lower price.

3.2.1 Description

The participants are a set of users with vehicles and a service provider. We assume that time is split into periods. The following three phases detailed below are executed in order by each vehicle during each period. At the start of a period, the vehicle generates fresh random tags for the period and registers commitments to hashed versions of these with the service provider (registration phase). Then, whenever the vehicle must emit a message containing an identifier during the period it will choose a new tag from its set of fresh tags. The tags are emitted in clear and the service provider records all tags v emitted by all vehicles together with the emission location l and a timestamp t, building a database containing a mixture of tuples (v, t, l) (driving phase). Finally, at the end of a period, each user initiates a protocol with the service provider in order to compute and settle the payable debts (reconciliation phase).

In the following, $[M]_d$ denotes a commitment to message M that can only be revealed with opening key d. Moreover, it is assumed to be a homomorphic commitment scheme, thus we have that:

$$[M_1]_{d_1} \cdot [M_2]_{d_2} = [M_1 + M_2]_{d_1+d_2}.$$

We further use $f_k(M)$ to denote a deterministic one-way function f that is parametrized by a key k. Note that knowing $f_k(M)$ and k does not allow one to retrieve M but only to compute a new hash matching $f_k(M)$.

Phase 1 - Registration. Each vehicle generates a set V of fresh tags $v_1, \ldots, v_n$ and a set of fresh keys $k^1, \ldots, k^s$ for f. It furthermore generates opening keys $dk^1, \ldots, dk^s$ and $dv_1^1, \ldots, dv_n^s$. It then forms s round packages

$$\mathcal{V} \to \mathcal{S} : \; r^i = \left(id, i, [k^i]_{dk^i}, [f_{k^i}(v_1)]_{dv_1^i}, \ldots, [f_{k^i}(v_n)]_{dv_n^i}\right)$$

consisting of the round number $i \in [1; s]$, a commitment to the round key k^i, and commitments to encryptions of the vehicle's tags under the round key. The vehicle sends the round packages to the server together with a fixed identifier id for the user, such as the vehicle's license plate.

Phase 2 - Driving. Each vehicle emits its tags $v_1, \ldots, v_n$ in random order along its route. The server records these tags along with the location l where it was emitted and a timestamp t.

Phase 3 - Reconciliation. The server starts the reconciliation phase by sending the list W of all m tags w_j in its database together with their associated cost c_j, *i.e.* W contains all tags emitted by all vehicles during the period. Then the vehicle computes C as the sum of the costs of its own tags contained in W and sends this back to the server

$$\begin{aligned}\mathcal{S} \to \mathcal{V}: &\quad W = \Big[(w_1, c_1), \ldots, (w_m, c_m)\Big] \\ \mathcal{V} \to \mathcal{S}: &\quad id, C\end{aligned}$$

The remaining part of the protocol consists of several rounds. For each round i, the vehicle generates opening keys $dc_1^i, \ldots, dc_m^i$. Then, it processes all pairs in W by hashing the tag w_j under its round key k^i and committing to the associated cost c_j using opening key dc_j^i. It permutes the pairs using a random permutation σ^i and sends the resulting list U^i to $\mathcal{S}$ together with its identifier. Then, the server decides to either verify that U^i is indeed the correct processing of W under k^i and $dc_1^i, \ldots, dc_m^i$ or to verify that the user has correctly calculated the cost C. In the former case it sends $b^i = 0$ to the vehicle and in the later case $b^i = 1$.

$$\begin{aligned}\mathcal{V} \to \mathcal{S}: &\quad id, U^i = \Big[\big(f_{k^i}(w_{\sigma^i(1)}), [c_{\sigma^i(1)}]_{dc_1^i}\big), \ldots, \big(f_{k^i}(w_{\sigma^i(m)}), [c_{\sigma^i(m)}]_{dc_m^i}\big)\Big] \\ \mathcal{S} \to \mathcal{V}: &\quad b^i \\ \mathcal{V} \to \mathcal{S}: &\quad \begin{cases} id, dk^i, dc_1^i, \ldots, dc_m^i & \text{if } b^i = 0 \\ id, dv_1^i, \ldots, dv_n^i, D^i & \text{if } b^i = 1 \end{cases}\end{aligned}$$

If $b^i = 0$ the server receives $dk^i, dc_1^i, \ldots, dc_m^i$ from the vehicle. It can then obtain k^i from r^i and verify that U^i correctly follows from W. If $b^i = 1$ the server receives $dv_1^i, \ldots, dv_n^i$ to open the commitments in r^i to obtain the hashed version of the vehicle's tags $f_{k^i}(v_1), \ldots, f_{k^i}(v_n)$. Knowing these it can pick out the pairs from U^i belonging to the vehicle (by the deterministic nature of f_{k^i}). It multiplies the cost commitments from these together and verifies that they indeed open to C under opening key D^i that is provided by the vehicle.

If the above mentioned checks pass for every round then the server accepts, the client is billed, and a new period begins. It can be argued that the probability of a cheating vehicle convincing the server of a false cost is 2^{-s}. This probability can hence be made arbitrary low by choosing a large enough number of rounds.

Spot checks. Note that there is no mechanism in the protocol described above that prevents vehicles from cheating by not emitting the tags they have committed to in order to reduce the price they have to pay. To address this the protocol suggests random spot checks. A spot check consists of an enforcement device that secretly collects identifying data about passing vehicles at locations where they are supposed to emit tags, for instance by taking a photograph of the license plate. This way it will obtain a database DB of tuples (id, v, l, t) each with an identity id, a tag v, a location l, and a timestamp t. Before the reconciliation phase the server will ask the vehicle about its identity id, challenge it with $\{(l, t) \mid (id, v, l, t) \in DB\}$, and *e.g.* fine the user if it fails to provide matching tags or the provided tags are not in the server's database. Since the location of the spot checks are assumed to be unknown to the vehicles, it can be argued that they do not know when they can safely avoid emitting a (valid) tag and hence must always do so when they are supposed to. Note that some privacy is leaked due to the spot checks; it is argued that this is at an acceptable level.

3.2.2 Privacy

The privacy definition for the VPriv scheme as stated in [BBP09] asks that the privacy guarantees from the system are the same as those of a system in which the server, instead of storing tuples (v, t, l), stores only tag-free path points (t, l). In other words, from the server's point of view, the tags might just as well be uncorrelated and random. This definition accounts for the fact that some privacy leaks are unavoidable and should not be blamed on the system. For instance, if one somehow learns that only a single vehicle was on a certain road at a particular

time, then that vehicle's tags can of course be linked to the tags emitted along the road at that time.

3.3 Formal Model

The process calculus used as input to the ProVerif tool is a variant of the applied pi calculus [AF01], a process calculus for formally modelling concurrent systems and their interactions. We here recall the basic ideas and concepts of this calculus that are needed for our analysis.

3.3.1 Messages

To describe messages, we start with a set of *names* used to identify communication channels and other atomic data, a set of *variables* $x, y, \ldots$ and a signature Σ formed by a finite set of *function symbols* $\mathsf{f}, \mathsf{g}, \ldots$ each with an associated arity. Function symbols are distinguished into two categories: *constructors* and *destructors*. We use standard notation for function application, *i.e.* $\mathsf{f}(M_1, \ldots, M_n)$. Constructors are used for building messages. Destructors represent primitives for taking messages apart and can visibly succeed or fail (while constructors always succeed). Messages $M, N, \ldots$ are obtained by repeated application of constructors on names and variables whereas a term evaluation D can also use destructors. The semantics of a destructor g of arity n is given by a set of rewrite rules of the form $\mathsf{g}(M_1, \ldots, M_n) \to M_0$ where $M_0, \ldots, M_n$ are messages that only contains constructors and variables. Given a term evaluation D, we write $D \Downarrow M$ when D can be reduced to M by applying some destructor rules.

In the following, we consider constructors to model commitments and the one-way function f. Since there is no destructor associated to f we have only one destructor whose semantics is given by the following rule:

$$\mathsf{open}(\mathsf{com}(x, y), y) \quad \to \quad x.$$

The applied pi calculus is quite general: it allows us, for instance, to model the homomorphism property of the commitment scheme by means of an equational theory containing, among others, the equation

$$\mathsf{com}(x_1, y_1) \times \mathsf{com}(x_2, y_2) = \mathsf{com}(x_1 + x_2, y_1 + y_2).$$

However, since ProVerif will not be able to reason with this equation, we will remove the homomorphic property in Section 3.5 and instead consider a simplified version of the protocol with no costs.

3.3.2 Processes

Processes are built from the grammar described in Figure 3.1, where M is a message, D is a term evaluation, n and c are names, x is a variable, and i is a positive integer.

The process "let $M = D$ in P else Q" tries to evaluate D; if this succeeds and if the resulting message matches the term M then the variables in M are bound and P is executed; if not then Q is executed. As explained in [Bla04], the process phase $i; P$ indicates the beginning of phase i. Intuitively, the process first executes phase 0, that is, it executes all instructions not under phase $i \geq 1$. Then, when changing from phase i to phase $i + 1$, all processes that have not reached a phase $i' \geq i + 1$ instruction are discarded and the instructions under phase $i + 1$ are executed.

The rest of the syntax is quite standard. To ease the presentation we will use tuples of messages, denoted by parentheses, while keeping the reduction rules for these tuples implicit.

$P, Q, R ::=$	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
new $n; P$	name restriction
let $M = D$ in P else Q	term evaluation
$\text{in}(c, M); P$	message input
$\text{out}(c, M); P$	message output
phase $i; P$	phase separation

Figure 3.1: Process grammar

$$
\begin{array}{l}
\mathsf{Vehicle}(id, l_1, l_2) \stackrel{\mathsf{def}}{=} \\
\quad \text{phase } 1; (*\ \textit{registration phase}\ *) \\
\quad \text{new } v_1;\ \text{new } v_2; \\
\quad \text{new } k;\ \text{new } dk;\ \text{new } dv_1;\ \text{new } dv_2; \\
\quad \text{out}(c, (id, \mathsf{com}(k, dk), \mathsf{com}(\mathsf{f}(v_1, k), dv_1), \mathsf{com}(\mathsf{f}(v_2, k), dv_2))); \\
\quad \text{phase } 2; (*\ \textit{driving phase}\ *) \\
\quad \text{out}(c, (l_1, v_1));\ \text{out}(c, (l_2, v_2, id)); \\
\quad \text{phase } 3; (*\ \textit{reconciliation phase}\ *) \\
\quad \text{in}(c, (x_1, x_2, x_3)); \\
\quad \text{out}(c, (id, \mathsf{f}(x_2, k), \mathsf{f}(x_1, k), \mathsf{f}(x_3, k))); \\
\quad \text{in}(c, y); \\
\quad \text{if } y = \mathsf{false} \text{ then } \text{out}(c, (id, dk)) \text{ else } \text{out}(c, (id, dv_1, dv_2))
\end{array}
$$

Figure 3.2: Illustrative vehicle process

We will omit "else Q" when Q is 0. In the remainder of the chapter, we use the more intuitive notation "if $M = N$ then P else Q" instead of "let $M = N$ in P else Q".

An *evaluation context* C is a process with a hole, built from $[\_]$, $C \mid P$, $P \mid C$ and new $n; C$. We obtain $C[P]$ as the result of filling the hole in C with P. A process P is *closed* if all its variables are bound through an input or a let construction.

The vehicle process. To illustrate the calculus used throughout this chapter, we give in Figure 3.2 a partial description of the vehicle process. It follows the description given in the previous section but is simplified in several aspects to keep this illustrative example as simple as possible. A more accurate model is described in Section 3.5.

Here we consider the case where a vehicle only has two tags v_1 and v_2, and where the reconciliation phase consists of only one round. We assume that during the driving phase the vehicle will visit only two locations and that the vehicle is spot checked at the second location. The vehicle receives a list of tags of size three (in reality, the length of the list is not known *a priori*), and instead of applying a random permutation, we only encode one particular permutation.

Semantics. The operational semantics of processes is essentially defined by two relations, namely *structural equivalence* and *reduction*. Structural equivalence, denoted by $\equiv$, is the smallest equivalence relation on processes that is closed under application of evaluation contexts and standard rules such as associativity and commutativity of the parallel operator. Moreover,

in order to deal with the phase construct, we also have structural rules for phases (see [BAF08]):

$$\begin{array}{rcl} \text{new } n; \text{phase } i; P & \equiv & \text{phase } i; \text{new } n\ ; P \\ \text{phase } i; (P \mid Q) & \equiv & \text{phase } i; P \mid \text{phase } i; Q \\ \text{phase } i; \text{phase } i'; P & \equiv & \text{phase } i'; P \quad \text{if } i < i' \end{array}$$

Reduction at phase i, denoted by $\rightarrow_i$, is the smallest relation closed under structural equivalence and application of evaluation contexts such that:

$$\begin{array}{ll} \text{RED I/O} & \text{phase } i; (\text{out}(c, M).Q \mid \text{in}(c, N).P) \rightarrow_i \text{phase } i; (Q \mid P\sigma) \\ \text{RED FUN 1} & \text{phase } i; \text{let } N = D \text{ in } P \text{ else } Q \rightarrow_i \text{phase } i; P\sigma \quad \text{if } D \Downarrow M \\ \text{RED FUN 2} & \text{phase } i; \text{let } N = D \text{ in } P \text{ else } Q \rightarrow_i \text{phase } i; Q \\ & \qquad \text{if there is no } M \text{ such that } D \Downarrow M \\ \text{REPL} & \text{phase } i; !P \rightarrow_i \text{phase } i; (P \mid !P) \end{array}$$

where σ is the substitution defined on the variables that occur in N and such that $M = N\sigma$. In case such a substitution does not exist, the resulting process will be $Q \mid \text{in}(c, N).P$ for the RED I/O rule and Q for the RED FUN 1 rule. We denote $\rightarrow = \bigcup_{i \geq 0} \rightarrow_i$ and we write $\rightarrow^*$ for the reflexive and transitive closure of reduction.

3.4 Privacy for Interactive Zero-Knowledge Protocols

We will define privacy using indistinguishability, which in turn will be formalized by a notion of equivalence. Equivalences have already been used to model privacy properties in formal analysis for *e.g.* vehicular mix-zones [DDS10] and electronic voting [KR05, DRS08]. The precise notion used is often *observational equivalence* but as we will explain, it happens that this notion is too strong to analyse interactive zero-knowledge protocols. So, we will rely on *trace equivalence* to formalize our notion of privacy in Section 3.4.2. However, the only equivalence relation supported by ProVerif is a stronger notion called *diff-equivalence*, and thus we explain in Section 3.4.3 how to use this tool to analyse trace equivalence-based properties.

3.4.1 Equivalences

One equivalence notion for formalizing indistinguishability is observational equivalence introduced in [Mil80, AF01]. Here we write $P \Downarrow_c$ when P can send an observable message on the channel c; that is, when $P \rightarrow^* C[\text{phase } i; \text{out}(c, M); Q]$ for some evaluation context C that does not bind c, some message M, some process Q, and some integer i.

Definition 3.4.1 (Observational equivalence)**.** Observational equivalence*, denoted* $\sim_o$*, is the largest symmetric relation* $\mathcal{R}$ *on closed processes* P *and* Q *such that* $P\ \mathcal{R}\ Q$ *implies:*

1. *if* $P \Downarrow_c$ *then* $Q \Downarrow_c$*;*
2. *if* $P \rightarrow P'$ *then there exists* Q' *such that* $Q \rightarrow^* Q'$ *and* $P'\ \mathcal{R}\ Q'$*;*
3. $C[P]\ \mathcal{R}\ C[Q]$ *for all evaluation contexts* C*.*

Intuitively, a context may represent an attacker, and two processes are observationally equivalent if they cannot be distinguished by any attacker at any step: every output step in an execution of process P must have an indistinguishable equivalent output step in the execution of process Q. If not then there exists a context that 'breaks' the equivalence.

In the case of privacy for the VPriv protocol, we will see that this notion is too strong (see the discussion in Section 3.4.2). Instead, we will use the notion of *trace equivalence* (also called *testing equivalence* in some other contexts [AG97]).

Definition 3.4.2 (Trace equivalence)**.** Trace equivalence $\sim_t$ *is the largest symmetric relation on closed processes* P *and* Q *such that for all evaluation contexts* C *we have* $C[P]\Downarrow_c$ *if and only if* $C[Q]\Downarrow_c$*.*

This is a strictly weaker notion than observational equivalence (see *e.g.* [CD09]) but intuitively it captures the equivalence upon which we can *a priori* hope to base our privacy property, as we explain below.

3.4.2 Formal Definition of Privacy

In our formal privacy definition we will assume that we have at least two honest vehicles called A and B. As we are interested in studying privacy guarantees for A, the process V_A for this vehicle will consist of all three phases of the protocol (registration, driving, and registration). We assume that vehicle A has three tags, one of which is emitted at one of the two locations $route_{left}$ and $route_{right}$, one which is 'leaked', *i.e.* given to the server along with the vehicle's identity to model the spot-check procedure, and one which is not emitted. On the other hand, vehicle B is only needed to counterbalance the effect of the tag emitted by A at a *route* location. Thus, we will consider a vehicle B that only executes its driving phase, denoted V_B^{dri} in the equivalence below, by emitting its tag at the *route* not visited by vehicle A.

We say that privacy holds if the following equivalence holds:

$$C_T\big[\,V_A(route_{left}) \mid V_B^{\mathsf{dri}}(route_{right})\big] \sim_t C_T\big[\,V_A(route_{right}) \mid V_B^{\mathsf{dri}}(route_{left})\big]$$

where C_T is an evaluation context modelling additional assumptions that may have to be made for the privacy property to hold (*e.g.* that the server is curious but otherwise assumed to be honest and following the protocol, or the existing of a trusted third party helping vehicles ensure that the list of tags received from the server contains tags from both vehicles). The next section presents the analysis we have performed, including the definition of the vehicles processes and the different contexts C_T within which we have performed the analysis.

Note that observational equivalence would be too strong for this property to hold. This is due to the interactive zero-knowledge subprotocol that occurs in the reconciliation phase. Consider the two slightly different processes $V_A(route_{left})$ and $V_A(route_{right})$ in our privacy definition and assume that the two processes have reached the reconciliation phase. At this point, the server will send a list of tags to vehicle A. Then one of the two processes, say the former one, will commit to a permuted list. To mimic this step, the latter process has also to commit to a permuted list. However, no matter what list it commits to, this list will not mimic the former process' list for either $b = 0$ or $b = 1$ because of the slight difference between them. In other words, the choice of a list to mimic the former process depends on the challenge bit b that has not yet been received from the server. Thus observational equivalence is impossible to achieve. However, moving from observation equivalence to trace equivalence allows us to choose the mimic trace only after the challenge bit has been learned. Intuitively, this captures privacy: if an attacker observes a trace of registration, tag emission and reconciliation, and then guesses that vehicle A took a particular route, then there is an equivalent trace where vehicle A takes a different route. The fact that we cannot specify the equivalent trace until we have seen the whole of the first trace does not seem to lead to any loss of privacy. In fact, from the definition of zero-knowledge protocols in the computational model [Gol01, §4] we see that the protocol is actually designed to support only trace equivalence and furthermore, that soundness contradicts observational equivalence.

3.4.3 Checking Privacy with ProVerif

The basic idea behind equivalence checking in ProVerif is to overlap the two processes that are supposedly equivalent, thereby forming a *biprocess* B. To achieve this, the syntax of ProVerif contains a $choice[M, M']$ operator which allows us to model a pair of processes that have the same structure and differ only in the choice of terms. Given a biprocess B, the process $P = fst(B)$ is obtained by replacing all occurrences of $choice[M, M']$ in B with M. Similarly, $Q = snd(B)$ is obtained by replacing $choice[M, M']$ with M'. When ProVerif is able to conclude positively on B, this implies that $P \sim_o Q$. However, ProVerif checks a stronger equivalence than observational equivalence and hence it fails on some simple examples of processes that are equivalent, but whose equivalence cannot be simulated by the moves of a single biprocess.

We will use two transformations in order to use ProVerif to check the trace equivalence defining our privacy property. The first arises from recent work which shows how to use ProVerif to prove observational equivalence for a wider class of processes [DRS08]. Additionally, we also transform our biprocess B into another biprocess B' that preserves the traces of each underlying process, *i.e.* $fst(B)$ and $fst(B')$ will produce the same traces, and likewise for $snd(B)$ and $snd(B')$. This ensures that our transformation preserves trace equivalence. In our case study this transformation consists of guessing b in advance and deadlocking the process if it later turns out that the guess was wrong.

3.5 Privacy Analysis

The purpose of our analysis is to investigate the privacy guarantees provided for an honest user in the VPriv protocol. We do not attempt to analyse whether users can cheat the server nor whether the server will accuse an honest user of cheating.

Section 3.5.1 contains a description of the simplifications we had to make in order to carry through the analysis in ProVerif. In Section 3.5.2 we describe our formal model of the VPriv protocol using the applied pi calculus from the previous section. We give the results of our analysis in Section 3.5.3.

3.5.1 Simplifications

The following simplifications were necessary in order to carry through the analysis in ProVerif.

Removing costs. In the extreme case where a unique price is used for every tag, the system cannot protect the privacy of users. It seems reasonable however, to assume that the information leaked by costs will in practice not affect the privacy of users. Forcing a uniform cost for every tag seems to be the only solution if we want to carry out our analysis with ProVerif. Furthermore, while we could model the homomorphic commitment scheme and its arithmetic properties by means of an equational theory, we know that ProVerif will not be able to deal with it properly in that it will not terminate. Thus, we remove prices and costs and proceed with a simplified version of the VPriv protocol. This change only affects the reconciliation phase where the list W sent by the server is now simply

$$\mathcal{S} \to \mathcal{V}: \quad W = \left[w_1, \ldots, w_m\right]$$

and the round subprotocol as described in Figure 3.3.

Fixing the length of W. It turns out that privacy can be violated if the list of tags sent by the server is blindly accepted by the vehicles without any scrutiny. Some sanity conditions must be fulfilled in order to guarantee privacy. Furthermore, implementing these sanity checks

$$
\begin{aligned}
&\mathcal{V} \to \mathcal{S}: \quad id, U^i = \left[f_{k^i}(w_{\sigma^i(1)}), \ldots, f_{k^i}(w_{\sigma^i(m)}) \right] \\
&\mathcal{S} \to \mathcal{V}: \quad b^i \\
&\mathcal{V} \to \mathcal{S}: \quad \begin{cases} id, dk^i & \text{if } b^i = 0 \\ id, dv_1^i, \ldots, dv_n^i & \text{if } b^i = 1 \end{cases}
\end{aligned}
$$

Figure 3.3: Reconciliation round protocol without cost.

$$
\begin{aligned}
&B_S(id^A, v_1^A, v_2^A, v_3^A, v_1^B, v_l, v_r) \stackrel{\mathsf{def}}{=} \\
&\quad \text{new } pc; \\
&\qquad \text{phase } 2; B_{\mathsf{dri}} \\
&\qquad \mid \text{phase } 3; B_{\mathsf{BB}} \\
&\qquad \mid !\ \text{new } k, dk, dv_1, dv_2, dv_3 \ \left(\text{phase } 1; V_{\mathsf{reg}}; \text{phase } 3; V_{\mathsf{rec}}^{b=0}\right) \\
&\qquad \mid !\ \text{new } k, dk, dv_1, dv_2, dv_3 \ \left(\text{phase } 1; V_{\mathsf{reg}}; \text{phase } 3; V_{\mathsf{rec}}^{b=1}\right)
\end{aligned}
$$

Figure 3.4: System Biprocess

together with the random permutation would lead us to consider a complex model that ProVerif is not able to handle. So instead, we fix *a priori* the length of the list expected by the vehicle to a size of three. This will allow us to easily encode the sanity checks and the random permutation, and despite its simplicity, still allow us to discover a number of issues to which attention should be paid when implementing the protocol. Note that with the sanity conditions discovered we can argue that fixing the length to three does not weaken the attacker.

3.5.2 Analysis Model

The model is represented by the biprocess B_S defined in Figure 3.4 and consisting of five parts: B_{dri}, V_{reg}, $V_{\mathsf{rec}}^{b=0}$, $V_{\mathsf{rec}}^{b=1}$, and B_{BB}. The first four of these together make up the behaviour of vehicle A and vehicle B. Using the choice operator the emitter biprocess B_{dri} outputs the tags of both vehicles while V_{reg} and the two V_{rec} are responsible for performing registration and reconciliation, respectively, for vehicle A. By splitting up vehicle A in this way we accurately model an unbounded number of reconciliation rounds while only emitting its tags once. The bulletin board B_{BB} is responsible for performing sanity checks on W. It receives a list of tags on a public channel and forwards the list to the V_{rec} biprocesses on the private channel pc an unbounded number of times if the checks succeed. Note that to avoid trivial false attacks, any checks against v_1^A and v_1^B must use the choice operator and hence the bulletin board is a biprocess. Finally, we use ProVerif's phases to orchestrate the processes so that they follow the order dictated by the protocol.

As discussed in Section 3.4, in order to establish the equivalence between the two cases, the selection of permutation for U^i depends upon the bit b^i that will be send by the server. We have two separate reconciliation processes $V_{\mathsf{rec}}^{b=0}$ and $V_{\mathsf{rec}}^{b=1}$ to model this. They guess that $b = 0$ and $b = 1$ will be sent, respectively, and permute accordingly. If the guess was correct the process proceeds as dictated by the protocol, otherwise it comes to a deadlock. Formally, let process P_{xyz} be defined by

$$P_{xyz} = \text{in}(s_1, \cdot); \text{out}\Big(c, \big(id^A, f(w_x, k), f(w_y, k), f(w_z, k)\big)\Big); \text{out}(s_2, \cdot); 0$$

which outputs the encrypted tags w_1, w_2 and w_3 permuted according to xyz. The initial input on s_1 is used to insure that only a single permutation is selected and the final output on s_2 to indicate that the output was completed. The $\cdot$ stands for any name never used after it is

$$
\begin{aligned}
&V_{\mathsf{rec}}^{b=0}(id^A, pc, k, dk) \stackrel{\mathsf{def}}{=} \\
&\quad \text{new } s_1, s_2; \\
&\quad \text{in}(pc, (w_1, w_2, w_3)); \\
&\qquad \text{out}(s_1, \cdot); 0 \\
&\qquad \mid P_{123} \mid P_{132} \mid P_{213} \mid P_{231} \mid P_{312} \mid P_{321} \\
&\qquad \mid \text{in}(s_2, \cdot); \text{in}(c, b);\ \text{if } b = 0 \text{ then out}(c, (id^A, dk))
\end{aligned}
$$

Figure 3.5: Reconciliation process for $b = 0$

$$
\begin{aligned}
&V_{rec}^{b=1}(id^A, pc, k, dv_1, dv_2, dv_3) \stackrel{\mathsf{def}}{=} \\
&\quad \text{new } s_1, s_2; \\
&\quad \text{in}(pc, (w_1, w_2, w_3)); \\
&\qquad \text{out}(s_1, \cdot); 0 \\
&\qquad \mid P_{123} \mid P_{132} \mid P_{213} \mid P_{231} \mid P_{312} \mid P_{321} \\
&\qquad \mid \text{in}(s_2, \cdot); \text{in}(c, b);\ \text{if } b = 1 \text{ then out}(c, (id^A, dv1, dv2, dv3))
\end{aligned}
$$

Figure 3.6: Reconciliation process for $b = 1$

bound. Using this process we then define $V_{\mathsf{rec}}^{b=0}$ as shown in Figure 3.5 and $V_{\mathsf{rec}}^{b=1}$ as shown in Figure 3.6. We note that because of the diff-equivalence that ProVerif is actually checking (see Section 3.4.3) it will only try to match the permutations at the same syntactical position. This means that we have to specify to ProVerif how permutations should be matched. For $V_{\mathsf{rec}}^{b=0}$ we can choose the same permutation in the two cases and hence no further modelling is needed and $V_{\mathsf{rec}}^{b=0}$ is actually just a process. However, this is not true for $V_{\mathsf{rec}}^{b=1}$ where we have to move the processes P_{xyz} around depending on which case we are in. We do this using the choice operator and hence $V_{\mathsf{rec}}^{b=1}$ is a biprocess. Let v_l be the tag emitted at $route_{left}$ and v_r the tag emitted at $route_{right}$. We have then chosen to arrange the permutations based on $w_1 = v_l$ and $w_3 = v_r$ and hence need to enforce this in the bulletin board.

3.5.3 Analysis Results

Unsurprisingly, it turns out that we have no privacy if W only contains tags of a single vehicle. It is necessary to ensure that the tags of both of the two honest vehicles are included in W, *i.e.* that W contains at least v_1^A (the tag emitted by vehicle A at its *route* location), and v_1^B (the tag emitted by the vehicle B at its *route* location). Actually, this is not sufficient since the server can still break privacy by sending a list with duplicates. An attack using this trick was reported by ProVerif.

With the above model B_S we performed several analyses by varying the sanity checks performed by the bulletin board. For the simplest case *without any checks* on W an attack is reported where arbitrary values are sent for $w_1 = w_2$ and w_3. This is a false attack caused by the way we match permutations in $V_{\mathsf{rec}}^{b=1}$. We can investigate the need for checks by removing all $V_{\mathsf{rec}}^{b=1}$. Then a real attack is reported: by sending arbitrary values for w_1 and w_2 and $w_3 = v_l$ the server can tell the cases apart when it sends $b = 0$.

In the case with W subject to *inclusion checks only* the attacker is allowed to choose w_2 but must send $w_1 = v_l$ (i.e. whichever tag was emitted in the left location) and $w_3 = v_r$ (whichever tag was emitted on the right). An attack is found when $w_2 = v_l$ by a comparison of the encrypted elements of U^i.

Finally, for the case with W subject to *inclusion checks and no duplicates* ProVerif is unable to conclude when *no duplicates* is interpreted as $w_2 \neq v_l \wedge w_2 \neq v_r$. However, if we interpret

this as $w_2 = v_2^A \vee w_2 = v_3^A$, i.e. rather than using an arbitrary tag not equal to v_l or v_r, the attacker must specifically use one of the unused registered tags, ProVerif is able to prove the equivalence and thus the privacy property for our model.

3.5.4 Evaluation

We evaluate first the VPriv protocol, then our analysis. Results on the privacy-preserving properties of the protocol are largely positive, at least in our abstract model. We discovered only privacy breaches that are possible for an active attacker who can tamper with the list, not for an 'honest but curious' attacker who merely inspects the protocol trace. We proposed some checks that could be made on the list W to thwart even an active attacker. The check for no duplicates is easy enough for a single vehicle to apply, but the check that the list really contains the tags of other vehicles is less easy and may require a trusted third party.

Turning to our analysis, it should be clear that a reasonable amount of work was required to develop an abstract model suitable for ProVerif whilst preserving the features of the protocol. However, it was not our aim to formalise the protocol just to exemplify the use of ProVerif but rather to push the boundaries of the tool in terms of protocol features. As such we have succeeded in identifying several features that a future version of the tool might handle better, namely lists, permutations, and homomorphic commitment schemes.

3.6 Conclusion

We have presented a privacy analysis of the VPriv scheme for anonymous location-based vehicular services. We have shown how a notion of trace equivalence captures the privacy notion the protocol is intended to provide, and have formally verified this property, albeit for an abstract model of the protocol. During our analysis we uncovered a number of areas where special attention needs to be paid when implementing such a protocol. We also introduced novel features into formal privacy modelling such as random list permutations and reasoning about interactive zero-knowledge protocols.

In future work we plan to investigate proofs of soundness for abstractions in the context of privacy properties, and apply our method to other privacy-enhancing protocols. In particular, it would be interesting to investigate a more general approach to reasoning about zero-knowledge protocols using the ProVerif tool set.

3.7 Bibliography

[ACRR10] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark D. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF'10)*, pages 107–121. IEEE Computer Society Press, 2010.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, New York, USA, 2001. ACM Press.

[AG97] Martín Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, Zurich (Switzerland), 1997. ACM Press.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

[BBP09] Andrew J. Blumberg, Hari Balakrishnan, and Raluca Popa. VPriv: Protecting privacy in location-based vehicular services. In *Proc. 18th Usenix Security Symposium*, 2009.

[BCdH10] Mayla Brusó, Konstantinos Chatzikokolakis, and Jerry den Hartog. Formal verification of privacy for RFID systems. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF'10)*. IEEE Computer Society Press, 2010.

[Bla04] Bruno Blanchet. *Cryptographic Protocol Verifier User Manual*, 2004. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[BMU08] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proc. Symposium on Security and Privacy (S&P'08)*, pages 202 –215. IEEE Computer Society Press, 2008.

[CD09] Véronique Cortier and Stéphanie Delaune. A method for proving observational equivalence. In *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 266–276, Port Jefferson, NY, USA, 2009. IEEE Computer Society Press.

[DDS10] Morten Dahl, Stéphanie Delaune, and Graham Steel. Formal analysis of privacy for vehicular mix-zones. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS'10)*, volume 6345 of *LNCS*, pages 55–70. Springer, 2010.

[DINI05] Marios D. Dikaiakos, Saif Iqbal, Tamer Nadeem, and Liviu Iftode. VITP: an information transfer protocol for vehicular computing. In *Proc. 2nd International Workshop on Vehicular Ad Hoc Networks (VANET'05)*, pages 30–39, 2005.

[DRS08] Stéphanie Delaune, Mark D. Ryan, and Ben Smyth. Automatic verification of privacy properties in the applied pi-calculus. In *Proc. 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08)*, volume 263 of *IFIP Conference Proceedings*, pages 263–278. Springer, 2008.

[Gol01] Oded Goldreich. *The Foundations of Cryptography*, volume 1. Cambridge University Press, 2001.

[KR05] Steve Kremer and Mark D. Ryan. Analysis of an electronic voting protocol in the applied pi-calculus. In *Proc. 14th European Symposium on Programming Languages and Systems (ESOP'05)*, volume 3444 of *LNCS*, pages 186–200. Springer, 2005.

[Law08] Nate Lawson. Highway to hell: Hacking toll systems. Presentation at Blackhat, 2008. Slides available from http://rdist.root.org/2008/08/07/fastrak-talk-summary-and-slides/.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Sta06] IEEE Standard. Trial-Use Standard for Wireless Access in Vehicular Environments – Security Services for Applications and Management Messages, Approved 8 June 2006.

Chapter 4

Universally Composable Symbolic Analysis

We consider a class of two-party function evaluation protocols in which the parties are allowed to use ideal functionalities as well as a set of powerful primitives, namely commitments, homomorphic encryption, and certain zero-knowledge proofs. We illustrate that with these it is possible to capture protocols for oblivious transfer, coin-flipping, and generation of multiplication-triple.

We show how any protocol in our class can be compiled to a symbolic representation expressed as a process in an abstract process calculus, and prove a general computational soundness theorem implying that if the protocol realises a given ideal functionality in the symbolic setting, then the original version also realises the ideal functionality in the standard computational UC setting. In other words, the theorem allows us to transfer a proof in the abstract symbolic setting to a proof in the standard UC model.

Finally, we show that the symbolic interpretation is simple enough in a number of cases for the symbolic proof to be partly automated using ProVerif.

4.1 Introduction

Giving security proof for cryptographic protocols is often a complicated and error-prone task. There is a large body of research aimed at doing something about this problem using methods from formal analysis [AR02, BPW03, CH06, CC08, CKW11]. This is interesting because the approach could potentially lead to automated or at least computer-aided (formal) proofs of security.

It is well known that the main difficulty with formal analysis is that it is only feasible when enough details about the cryptographic primitives have been abstracted away, while on the other hand this abstraction may make us "forget" about issues that make an attack possible. One solution to this problem is to show once and for all that a given abstraction is *computational sound*, which loosely speaking means that for any protocol, if we know there are no attacks on its abstract *symbolic* version then this (and some appropriate complexity assumption) implies there are no attacks on the original *computational* version. Or, in other words, that intuitively the symbolic adversary is as powerful as the computational adversary in the sense that his ability to distinguish in the real-world model is not greater than his ability to distinguish in the symbolic model. Such soundness theorems are known in some cases (see related work), in particular for primitives such as public-key encryption, symmetric encryption, digital signatures, and hash functions.

Another issue with formal analysis is how security properties should be specified. Traditionally this has been done either through trace properties or "strong secrecy" where two instances

of the protocol running on different values are compared to each other[1]. This approach has carried on to work on computational soundness where results are known for security properties such as authenticity and key secrecy. On the other hand, the cryptographic community has long recognised the usefulness of the simulation-based approach, not least when analysing protocols where the players take inputs from the environment.

Finally, making protocol (and in particular system) analysis feasible in general requires some way of breaking the task into smaller components which may be analysed independently. While also this has been standard in the cryptographic community for a while, in the form of eg. the UC framework [Can01, Can05], it has not yet received much attention in the symbolic community (but see [CH06] for an exception).

4.1.1 Our Results

In this chapter we make progress on expanding the class of protocols for which a formal analysis can be used to show security in the computational setting. We are particularly interested in two-party function evaluation protocols and the primitives used by many of these, namely homomorphic public-key encryption, commitments, and certain zero-knowledge proofs. We aim for proofs of UC security against an active adversary and where one of the parties may be (statically) corrupted.

Protocol model. We make some assumptions on the form of protocols. Besides the above primitives protocols are also allowed to use ideal functionalities and communicate over authenticated channels. We put some restrictions on how the primitives may be used. First, whenever a player sends a ciphertext it must be accompanied by a zero-knowledge proof that the sender knows how the ciphertext was constructed: if the ciphertext was made from scratch then he knows the plaintext and randomness used, and if he constructed it from other ciphertexts using the homomorphic property then he knows randomness that "explains" the ciphertext as a function of that randomness and ciphertexts that were already known. We make a similar assumption on commitments and allow also zero-knowledge proofs that committed values relate to encrypted values in a given way. Second, we assume that honest players use the primitives in a black-box fashion, ie. an honest player can run the protocol using a (private) "crypto module" that holds all his keys and handles encryption, decryption, commitment etc. This means that all actions taken by an honest player in the protocol may depend on plaintext sent or received but not, for instance, on the binary representation of ciphertexts. We emphasise that we make no such restriction on the adversary.

We believe that the assumptions we make are quite natural: it is well known that if a player provides input to a protocol by committing to it or sending an encryption then we cannot prove UC security of the protocol unless the player proves that he knows the input he provides. Furthermore, active security usually requires that players communication over authenticated channels and prove that the messages they send are well-formed. We stress, however, that our assumptions do not imply that an adversary must be semi-honest; for instance, our model does not make any assumptions on what type and relationship checks the protocol must perform, nor on the randomness distributions used by a corrupted player.

Security properties. As in the simulation-based paradigm we use *ideal functionalities* and *simulators* to specify and prove security properties. More concretely, we can express all three

[1] For strong secrecy one runs the same protocol on two fixed but different inputs (or with one instance patched to give an independent output) and then ask if it is possible to tell the difference between the two executions. This can for instance be used to argue that a key-exchange protocol is independent of the exchanged key given only the transmitted messages.

entities in our model and say that a *protocol ϕ is secure* (with respect to the ideal functionality $\mathcal{F}$) if no adversary can tell the difference between interacting with ϕ and interacting with $\mathcal{F}$ and simulator Sim[2], later written $\phi \sim \mathcal{F} \diamond Sim$ for concrete notions of indistinguishability. When this equivalence is satisfied we also say that the protocol *(UC) realises* (or *implements*) the ideal functionality.

We require that ideal functionalities only operate on plain values and do not use cryptography, and as with protocols we assume that simulators only use the primitives and their trapdoors through a crypto-module.

Proof technique. Our main result is quite simple to state on a high level: given a protocol ϕ, ideal functionality $\mathcal{F}$, and simulator Sim, we show how these may be compiled to symbolic versions such that if we are given a proof *in the symbolic world* that ϕ realises $\mathcal{F}$ then it follows that ϕ realises $\mathcal{F}$ *in the usual computational world* as well (assuming the crypto-system, commitment scheme, and zero-knowledge proofs used are secure). As usual for UC security, we need to make a set-up assumption which in our case amounts to assuming a functionality that initially produces reference strings for the zero-knowledge proofs and keys for the crypto-system.

We arrive at our result as follows: we first define a simple programming language for specifying protocols on a rather high and abstract level. The class of protocols we consider is then defined as whatever can be described in this language. More formally the language can talk about a set of entities that interact and we use the name *system* as a generic term for such a set of entities. In particular, we can talk about a system triple $\left(Sys^{\mathcal{H}}_{real}\right)_{\mathcal{H}}$ for $\mathcal{H} \in \{AB, A, B\}$ modelling the behaviour and components of a protocol ϕ, but also a triple $\left(Sys^{\mathcal{H}}_{ideal}\right)_{\mathcal{H}}$ that models $\mathcal{F}$ running together with a simulator. We denote the former a *real protocol* and the latter an *ideal protocol*.

We then define three different ways of interpreting such systems:

- *Real-world interpretation* $\mathcal{RW}(Sys)$: Assuming concrete instantiations of the cryptographic primitives this interpretation produces from system Sys a set of interactive Turing machines that fits in the usual UC model. For instance, $\mathcal{RW}(Sys^{AB}_{real})$ contains two ITMs M_A, M_B executing the player programmes of ϕ, while $\mathcal{RW}(Sys^{AB}_{ideal})$ contains $M_{\mathcal{F}}, M_{Sim}$ respectively executing the ideal functionality and the simulator.
- *Intermediate interpretation* $\mathcal{I}(Sys)$: This interpretation also produces a set of ITMs fitting into the UC model, but does not use concrete cryptographic primitives. Instead we postulate an ideal functionality that receives all calls to cryptographic functions and returns handles to objects such as encrypted plaintexts while storing these plaintexts in a restricted global memory. Players then send such handles instead of actual ciphertexts and commitments. A new component of this interpretation is that the adversary is now also given an operation module through which he is forced to launch his attack.
- *Symbolic interpretation* $\mathcal{S}(Sys)$: This interpretation closely mirrors the intermediate interpretation but instead produces a set of *processes* described in a well-known process calculus. This forms our symbolic model.

Having defined these interpretations we define notions of equivalence of systems in each representation: $\mathcal{RW}(Sys_1) \overset{c}{\sim} \mathcal{RW}(Sys_2)$ means that no polynomial time environment can distinguish the two cases given only the public and corrupted keys, and may for instance be used to capture that a protocol UC-securely realises $\mathcal{F}$ in the standard sense; for the intermediate world $\mathcal{I}(Sys_1) \overset{c}{\sim} \mathcal{I}(Sys_2)$ means the same but now in the global memory hybrid model; finally,

[2]Intuitively, the simulator is used to capture the "unimportant" differences between the two settings (e.g. that cryptography is used in the former but not the latter) and to interpret the actions of the adversary relative to the ideal functionality.

$\mathcal{S}(Sys_1) \overset{s}{\sim} \mathcal{S}(Sys_2)$ means the two processes are *observationally equivalent* in the standard symbolic sense.

We prove two soundness theorems stating first, that $\mathcal{I}(Sys_1) \overset{c}{\sim} \mathcal{I}(Sys_2)$ implies $\mathcal{RW}(Sys_1) \overset{c}{\sim} \mathcal{RW}(Sys_2)$ and second, that $\mathcal{S}(Sys_1) \overset{s}{\sim} \mathcal{S}(Sys_2)$ implies $\mathcal{I}(Sys_1) \overset{c}{\sim} \mathcal{I}(Sys_2)$, so that in order to prove UC security of a protocol[3] in our class it is now sufficient to show equivalence in the symbolic model and this is the part we can partly automate[4] using the ProVerif tool [BAF05].

Finally, we note that in some cases (in particular when both players are honest) it is possible to use a standard simulator construction and instead check a different symbolic criteria along the lines of previous work [CH06]. This removes the manual effort required in constructing simulators.

Analysis approach. Given the above, a protocol ϕ in our class may hence be analysed in our framework as follows:

1. formulate ϕ and its ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_n$ in our model and language
2. likewise formulate the target ideal functionality $\mathcal{G}$ and simulator *Sim*
3. let $\left(Sys^{AB}_{real}, Sys^{A}_{real}, Sys^{B}_{real}\right)$ and $\left(Sys^{AB}_{ideal}, Sys^{A}_{ideal}, Sys^{B}_{ideal}\right)$ be respectively the real protocol composed of ϕ and $\mathcal{F}_1, \ldots, \mathcal{F}_n$ and the ideal protocol composed of $\mathcal{G}$ and *Sim*; then show in the symbolic model, eg. using ProVerif, that $\mathcal{S}(Sys^{\mathcal{H}}_{real}) \overset{s}{\sim} \mathcal{S}(Sys^{\mathcal{H}}_{ideal})$ holds in all three cases
4. use the soundness theorem to conclude that $\mathcal{RW}(Sys^{\mathcal{H}}_{real}) \overset{c}{\sim} \mathcal{RW}(Sys^{\mathcal{H}}_{ideal})$, and in turn that ϕ realise $\mathcal{G}$ using simulator *Sim*

Note that as usually in the UC framework we only need to consider one session of the protocol since the compositional theorem guarantees that it remains secure even when composed with itself a polynomial number of times. Note also that we may apply our result to a broader class of protocols through a hybrid-symbolic approach where the protocol in question is broken down into several sub-protocols and ideal functionalities analysed independently either within our framework or outside in an ad-hoc setting (possibly using other primitives) as outlined in Section 4.8.1.

We have tried to make the models suitable for automated analysis using current tools such as ProVerif, and although our approach requires manual construction of a simulator for the symbolic version of the protocol, this is usually a very simple task. As a case study we in Section 4.7 carry out an analysis of the oblivious transfer protocol from [DNO08].

4.1.2 Related Work

The main area of related work is *computational soundness* which we go into detail with below, focusing in particular on three lines of closely related work. We refer to [CKW11] for an in-depth survey of this area.

We also mention the area of *symbolic modelling of security properties* using the simulation-based paradigm. To the best of our knowledge this paradigm has only received little attention in

[3]Note that we actually prove a slightly stronger result than what is needed to show UC security: it would have been enough to show that if a real protocol realises an ideal functionality in the intermediate model then it also does so in the real-world model. Concretely, we could have avoided defining a real-world interpretation of an ideal protocol. As an added bonus, this slightly stronger result means that the soundness theorem could also be used to give computational assurance to analyses where two instances of the real protocol (eg. running on different fixed inputs) are compared.

[4]Obviously, the symbolic equivalence could also be proved by hand or using some other tool. We have chosen ProVerif here for its resolving power and wide-spread acceptance in the symbolic community. Furthermore, if better tool support arises for a different symbolic model then the first part of our soundness result could of course be re-used.

the symbolic community, yet seems natural when analysing function evaluation protocols such as oblivious transfer. In particular, most symbolic models do either not follow this paradigm or do not give the simulator special powers (such as trapdoors).

Finally, there is also a large body of work on the *direct approach* where the symbolic model is altogether avoided but instead used as inspiration for creating a computational model easier to analyse. This line of work includes [Bla08, MRST06, DDMR07] and while it is more expressive than the symbolic approach we have taken here, our focus has been on abstracting and automating as much as possible.

Computational soundness. The line of work started by Backes et al. [BPW03] and known as "the BPW approach" gives an ideal cryptographic library based on the ideas behind abstract Dolev-Yao models. The library is responsible for all operations that players and the adversary want to perform (such as encryption, decryption, and message sending) with every message being kept in a database by the library and accessed only through handles. Using the framework for reactive simulatability [PW01] (similar to the UC framework) the ideal library is realised using cryptographic primitives. This means that a protocol may be analysed relative to the ideal library yet exhibit the same properties when using the realisation instead. The original model supporting nested nonce generation, public-key encryption, and MACs was later been extended to support symmetric encryption [BP04] and a simple form of homomorphic threshold-encryption [LN08] allowing a single homomorphic evaluation. The approach has also been used to analyse protocols for trace-based security properties such as authentication and key secrecy [BP03, BP06].

Comparing our work to the BPW approach we see that the operation modules and global memory functionality of our intermediate model correspond to the ideal cryptographic library, and the real-world operation modules to the realisation. In this light Lemma 4.5.4 and 4.5.5 form our realisation result[5]. The difference lies in the supported operations: namely our more powerful homomorphic encryption and simulation operations – the former allows us to implement several two-party functionalities while the latter allows us to express simulators for ideal functionalities within the model. This not only allows us to capture a different class of security properties[6] (such as the standard assumptions on oblivious transfer with static corruption) but also to do modular and hybrid-symbolic analysis. The importance of this was elaborated on in [Can08].

The next line of work closely related is that started by Canetti et al. in [CH06] and building on [MW04, BPW03] but adding support for modular analysis. They first formulate a programming language for protocols using public-key encryption and give both a computational and symbolic interpretation. They then give a mapping lemma showing that the traces of the two interpretation coincide, ie. the computational adversary can do nothing that the symbolic adversary cannot also do (except with negligible probability). While this only shows soundness of trace properties they are then able to lift this to indistinguishability properties for two special cases and give symbolic criteria for realising authentication and key-exchange functionalities. Moreover, they use ProVerif to automate the analysis of the original Needham-Schroeder-Lowe protocol (relative to authenticity) and two of its variants (relative to key-exchange). Later work [CG10] again targets key-exchange protocols but adds support for digital signatures, Diffie-Hellman key-exchanges, and forward security under adaptive corruption.

[5] Note that we put some requirements on the use of our "library" by demanding that protocols are well-formed; the BPW library works for an environment by instead putting these requirements in the code of the library.

[6] In principle the BPW model could be used as a stepping-stone to analyse cases where the simulator may simply run the protocol on constants. However, the simulator is sometimes required to use trapdoors in order to extract information needed to simulate an ideal functionality in the simulation-based paradigm. These cases cannot be analysed with the operations of the BPW model.

Most important, our approach has been that of not fixing the target ideal functionalities but instead letting it be expressible in the model (along with the realising protocol and simulator). Hence it is relatively straight-forward to analyse protocols realising other functionalities than what we have done here, whereas adapting [CH06] to other classes of protocols requires manually finding and showing soundness of a symbolic criteria. It is furthermore not clear which functionalities may be captured by symbolic criteria expressed as trace properties and strong secrecy. In particular, the target functionalities of [CH06] and [CG10] do not take any input from the players nor provide any security guarantees when a player is corrupt, and hence the criteria do not need to account for these case. Again we also show soundness for a different set of primitives.

The final line of related work is showing soundness of indistinguishability-based (instead of trace-based properties). This was started by Comon-Lundh et al. in [CC08] and, unlike the two lines of work mention above, aims at showing that if the symbolic adversary cannot distinguish between two systems in the symbolic interpretation then the computational adversary cannot do so either for the computational interpretation. [CC08] showed this for symmetric encryption and was continued in [CHKS12] for public-key encryption and hash functions.

Our work obviously relates in that we are also concerned about soundness of indistinguishability. Again the biggest difference is the choice of primitives, but also that our framework seems more suitable for expressing ideal functionalities and simulators: although mentioned as an application, their model do not appear to be easily adapted to capturing the typical structure of a composable analysis framework such as the UC framework (private channels are not allowed for instance, see also [Unr11]). And while their result may be used as a stepping stone they do not provide the essential simulator operations. To this end the result is closer to what might be achieved through the BPW approach. Note that the work in [CHKS12] does not require computable parsing (as we do through the NIZK proofs). However, for secure function evaluation in the simulation-based paradigm some form of computational extraction is typically required in general.

The work in [BMM10] is also somewhat related in that they also aim at analysing secure function evaluation, namely secure multi-party computations (MPC). However, they instead analyse protocols using MPC as a primitive whereas we are interested in analysing the (lower-level) protocols realising MPC. Moreover, they are again limited to trace properties.

Symbolic modelling of security properties. Most related work in the huge area of symbolic modelling of security properties (without computational soundness) is mainly focused on either trace-based properties or notions related to *strong secrecy*; so far the simulated-based paradigm has not gained much popularity. Delaune et al. [DKP09] show that the paradigm may be expressed in the applied-pi calculus and compare different instantiations including the UC framework[7]. From this perspective our model of the UC framework is simple and does not aim to capture as many aspects. On the other hand we give a computational interpretation and soundness result. More generally, to the best of our knowledge this is also the first work capturing secure functionalities such as oblivious transfer under corruption in a symbolic setting; expressing the security requirements for this is natural in the simulation-based paradigm but it is much less clear how this can be captured using trace-based properties or even strong secrecy[8].

[7]The computationally equivalent comparison was done in [DKMR05] from which we also took some inspiration when formulating our model.

[8]For oblivious transfer protocols we typically require two properties regarding the secrecy of the involved inputs: (i) even a corrupt sender does not learn the receiver's choice bit b; and (ii) even a corrupt receiver only learns the message x_b that he is asking for and nothing about x_{1-b}. While it might be possible to capture these using strong secrecy for the cases where both players or only the receiver is honest, it is less clear how to do this when only the sender is honest; we cannot for instance use $S(x_0, x_1) \sim S(0, x_1)$ to capture that x_0 should

Note that (randomised) oblivious transfer was expressed and analysed symbolically in the probabilistic applied-pi calculus in [GPT07] but not using the simulated-based paradigm and hence only for the case where both players are honest.

4.1.3 Organisation

The rest of the chapter is organised as shown next. As a reading hint, Section 4.2 should be read before later sections. Readers coming from the cryptographic community may then continue to read the chapter in the given order, following a "top-down" approach of progressively removing cryptography and bitstrings, and ending up with an highly idealised model. On the other hand, readers coming from the symbolic community may instead choose a "bottom-up" approach starting with the symbolic or intermediate interpretation and then replace the ideal cryptography with concrete schemes afterwards.

Section 4.2 specifies our protocol class including the interface of the crypto black-boxes, dubbed *operation modules*. It then introduces a simple programming language and illustrates how an oblivious transfer, a commitment, a coin-flipping, and a triple-generation protocol may be expressed, as well as their ideal functionalities and suitable simulators.

Section 4.3 gives the preliminaries for the real-world interpretation, ie. it introduces our computational setting in the form of the UC framework, and defines the assumptions we make on the cryptographic primitives.

Section 4.4 gives the *real-world interpretation* of a system in terms of the UC framework and the primitives. This amounts to specifying how protocols are executed and implementing the operation modules.

Section 4.5 gives the *intermediate interpretation* of a system still in terms of the UC framework but this time using an ideal global memory instead of the primitives. The soundness theorem is then shown by introducing the concept of a *translator* that maps messages between the two interpretations; the translator is in fact just a standard UC simulator but we want to avoid overloading this name.

Section 4.6 gives the *symbolic interpretation* as an abstraction of the intermediate model using a dialect of the applied-pi calculus. Soundness of symbolic indistinguishability is a simple result given the abstract nature of the intermediate model.

Section 4.7 shows how the oblivious transfer protocol we use as a running example may be analysed using ProVerif. Although this is not fully automated due to the nature of the tool, we instead give a systematic approach for massaging the processes of the symbolic interpretation to fit the tool.

Finally, in Section 4.8 we give a few remarks on possible extensions and future work.

be kept secret because this puts an assumption on the behaviour or the corrupted player (that he will ask for x_1) and would not hold in the valid case where he asks for x_0. From the simulation-based point of view, what is missing is a simulator and ideal functionality that during the protocol execution can decide which x_b he is asking for and then release no other information.

4.2 Protocol Model

This section introduces the class of protocols in consideration and for which the soundness result holds. We use the oblivious transfer (OT) protocol from [DNO08] as a motivating example of the general structure and the available primitives, and define two kinds of systems of programmes, namely *real protocols* and *ideal protocols*, respectively describing the actual protocol and its abstract behaviour.

Ending the section we give several examples of what is captured by our protocol class: besides the OT protocol, we also give a coin-flip (CF) protocol with a commitment sub-protocol, and a multiplication triple generation protocol used in secure multi-party computation. We give the corresponding ideal functionalities for all four protocols as well as suitable simulators.

4.2.1 Motivating Example

We introduce our protocol class by way of an running example. Consider the OT protocol from [DNO08] shown in Figure 4.1 with the sender programme on the left and the receiver programme on the right. Note that both agents know encryption key ek_R, the receiver knows the corresponding decryption key dk_R, and both agents know commitment key ck_S. Also, the sender expects values x_0, x_1 and the receiver bit b from the environment. Only the receiver sends back an output to the environment, namely x_b.

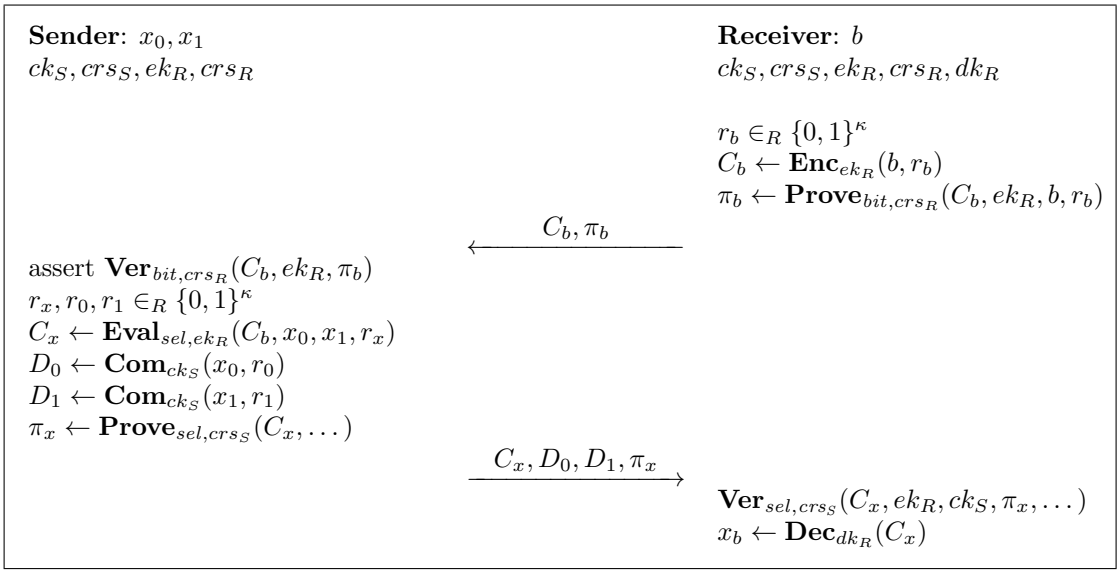

Figure 4.1: The OT protocol of [DNO08] in the original notation

In the first step of the protocol the receiver encrypts his bit b under his encryption key. When the sender next uses C_b to form an encyption C_x of either x_0 or x_1 it is critical for the security of the protocol that she ensures that the plaintext of C_b is really a bit: if the receiver sends an encryption of e.g. 2 then he would learn both x_0 and x_1 in the case where these are bits. The proof π_b ensures that C_b is really an encryption of a bit.

In the second step the sender uses the homomorphic properties of the encryption scheme to form C_x from C_b, x_0, and x_1. The expression she is evaluating is

$$sel(\alpha; \beta_0, \beta_1) \doteq \overline{\alpha} \cdot \beta_0 + \alpha \cdot \beta_1$$

where $\overline{\alpha} \doteq 1 - \alpha$ denotes negation if α is a bit[9]. On the receiver side a proof is needed to ensure correctness of the protocol in the sense that the sender combined the ciphertexts as she was supposed to. As pointed out in [DNO08] it is also needed to obtain composable security guarantees. Hence, the sender also commits to inputs x_0 and x_1 and forms a proof π_x that C_x was obtained by expression *sel* using C_b and the values in D_0 and D_1 as inputs.

Finally, in step three the receiver verifies the proof and decrypts C_x to obtain x_b; this is given as the output of the protocol to the environment.

The above OT protocol serves as a motivator for our choice of protocol class. The supported cryptography is commitments and homomorphic encryption with a fixed set of keys, and all common reference strings, public encryption keys, and commitment keys are honestly generated and known to everyone. We furthermore assume that commitments and encryptions are annotated with the public components needed to check their associated proofs, in particular the encryption and commitment keys[10]. In summary, we assume that commitments and encryptions always are of the following "package" forms:

$$\begin{aligned} \textit{commitment package:} &\quad [\texttt{comPack} : D, ck, \pi_U, crs] \\ \textit{encryption package:} &\quad [\texttt{encPack} : C, ek, \pi_T, crs] \\ \textit{evaluation package:} &\quad [\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs] \end{aligned}$$

and otherwise treated as garbage, resulting in abortion. We shall through out the chapter use $\mathcal{D}$ to range over commitment packages and $\mathcal{C}$ to range over both encryption and evaluation packages. This leads to a protocol class parameterised by a finite domain of values, two finite sets of types $\{T_i\}_i$, $\{U_j\}_j$, and two finite sets of arithmetical expressions $\{e_k\}_k \subseteq \{f_\ell\}_\ell$. Here, as in the rest of the chapter we shall often assume that the expressions are over four variables (mathcing the values in C_1, C_2, D_1, and D_2 in the evaluation package above) to simplify the presentation.

To fit into our protocol class and analysis framework we hence need to formulate the example OT protocol as shown in Figure 4.2 using operations encrypt, verEncPack etc. introduced below. The supported types are $T = bit$ and $U = dom$, where $bit = \{0, 1\}$ and dom is some plaintext space. The supported expression is $e = f = sel$ as defined above. We see that the biggest change is the use of packages instead of simple commitment, encryptions, and NIZK proofs. Note that the two commitments are now implicitly created through the eval_e instruction; we also allow the explicit creation of type annotated commitments but did not need this here.

A given protocol is analysed relative to a specification in the form of an ideal functionality. Since we concentrate on two-party protocols against active adversary with static corruption capabilities we basically have (up to) three scenarios to consider: when both player A and player B are honest, when only player A is honest, and when only player B is honest. For each of these scenarios we may ask if the adversary is able to tell if it is interacting with the honest players or with the ideal functionality combined with a simulator. For instance, to analyse the OT protocol with players S and R we ask if the adversary can tell the difference between interacting with S and R, or with the ideal functionality $\mathcal{F}_{OT}$ that is simply given the inputs (x_0, x_1 and b) and returns the correct output (x_0 if $b = 0$ and x_1 otherwise) to the receiver.

[9]Note that *sel* may be written as $\beta_0 + \alpha \cdot (-1) \cdot \beta_0 + \alpha \cdot \beta_1$ and requiring only a somewhat-homomorphic encryption scheme.

[10]These annotations are without loss of generality but means that we can talk about well-formed messages independent of the protocol.

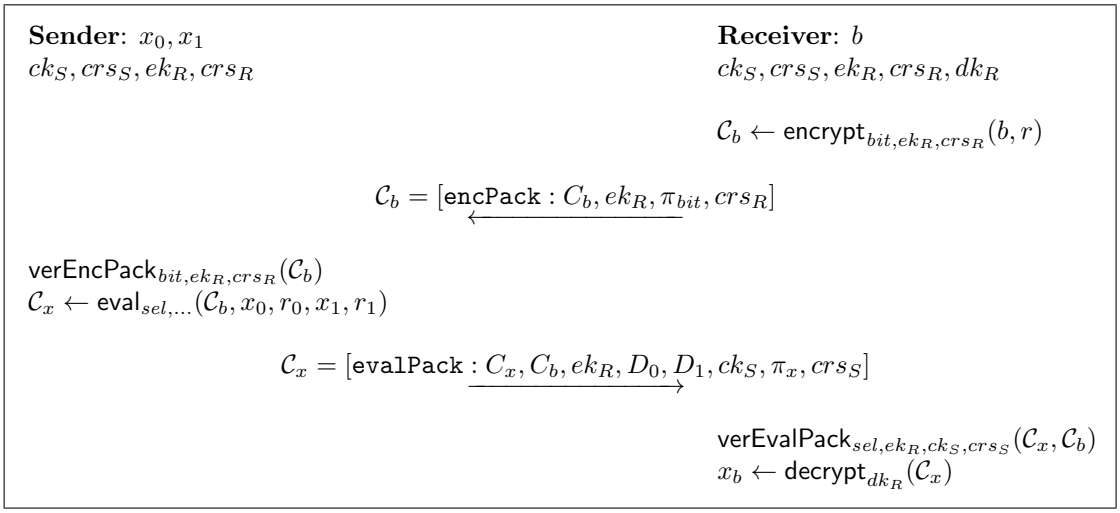

Figure 4.2: The OT protocol of [DNO08] in our annotated notation

4.2.2 Systems

We use *system* as a generics term for grouping a set of programmes $P_1, \ldots, P_n$ intended to be executed concurrently. We shall use $\diamond$ as in

$$Sys \doteq P_1 \diamond \cdots \diamond P_n$$

to denote the composition of such programmes into a system *Sys* connected through a set of directional *plain* and *crypto ports*. We shall also use $\diamond$ to combine systems.

We put a few requirements on a system for it to be well-formed. Firstly, we require that every port is used as an input port by at most one programme, and likewise as an output port by at most one programme (by a later restriction no programme can use a port both for input and output). Ports not used in both directions are dubbed *open* and accessible to the environment. Secondly, in our systems at most two programmes are labelled as being *cryptographic*, intuitively the programme representing player A and B (or their simulators).

4.2.3 Programmes

A *programme* P is structured as a constant number of input-process-output cycles and is specified over a fixed set of value symbols $\mathbb{V}$, a fixed set of constant symbols $\mathbb{C}$, a set of randomness symbols $\mathbb{R}_P$, a set of variables $\mathbb{X}_P$, a set of allowed operations $\mathbb{O}_P$ to be specified below, and a set of input and output ports respectively $\mathbb{P}_P^{in}$ and $\mathbb{P}_P^{out}$. Every processing of an input is done by a combination of operations from $\mathbb{O}_P$, through references that are substituted into the variables[11].

To describe programmes we may introduce a *simple programming language*. Consider for instance the OT sender and receiver from above; in our simple language they may be expressed as in Figure 4.3 with the sender on the left and the receiver on the right. We see that the receiver first listens on port in_{OT}^R. When an input arrives on this port it names it b, checks that it is

[11] We shall not dwell too much over the difference between variables and references here but return to it briefly when giving the computational semantics later. Note however, that the use of references allows us to give a precise specification of the operations a programme may perform, which is central to the later soundness results.

$P^S_{OT} \doteq \text{input}_\emptyset[\mathit{receive}_{RS} : c_b];$
if $\mathsf{verEncPack}_{bit,R,R}(c_b)$ then
output$[\mathit{out}^S_{OT} : \texttt{getInput}];$
input$_\emptyset[\mathit{in}^S_{OT} : x_{01}];$
if $\mathsf{isPair}(x_{01})$ then
let $x_0 \leftarrow \mathsf{first}(x_{01});$
let $x_1 \leftarrow \mathsf{second}(x_{01});$
if $\mathsf{isValue}(x_0)$ then
if $\mathsf{isValue}(x_1)$ then
let $c_x \leftarrow \mathsf{eval}_{sel,R,S,S}(c_x, x_0, r_0, x_1, r_1);$
output$[\mathit{send}_{SR} : c_x];$
stop

$P^R_{OT} \doteq \text{input}_\emptyset[\mathit{in}^R_{OT} : b];$
if $\mathsf{inType}_{bit}(b)$ then
let $c_b \leftarrow \mathsf{encrypt}_{bit,R,R}(b, r);$
output$[\mathit{send}_{RS} : c_b];$
input$_\emptyset[\mathit{receive}_{SR} : c_x];$
if $\mathsf{verEvalPack}_{sel,R,S,S}(c_x, c_b)$ then
let $x_b \leftarrow \mathsf{decrypt}_R(c_x);$
output$[\mathit{out}^R_{OT} : x_b];$
stop

Figure 4.3: Player programme P^S_{OT} for sender (left) and P^R_{OT} for receiver (right)

a bit using $\mathsf{inType}_{bit}(b)$[12], encrypts it using $\mathsf{encrypt}_{bit,R,R}(b, r)$[13], and sends the encryption on port send_{SR}; it then starts to listen on port $\mathit{receive}_{SR}$. Figure 4.4 shows the flow of the entire protocol.

Note that the sender programme P^S_{OT} is expressed in what may be considered an atypical manner where it first receives an encryption c_b from the receiver and then asks the environment for its input by sending a `getInput` constant on the open port out^S_{OT} to the environment. Upon receiving its input x_{01} it only then replies with c_x to the receiver. Expressing the sender this way the system is "non-losing" despite using only "simple" programmes as discussed next; if desired the sender may easily be patched to fit with the more typical notation.

Another thing to note about the language is that the input command input$_\mathcal{P}[p : x]$ is specified with a set of ports $\mathcal{P}$. The informal semantics is that the programme is also listening on all ports in this set; however, an input on these will result in the programme aborting. The motivation for having these is that the soundness result in Section 4.6.5 requires that systems are *non-losing*, in the sense that whenever a programme sends a message on a closed port p the receiving programme must also be listening on p. This property is easily satisfied by having every programme listen to all of its input ports at every programme point. However, doing this may also complicate analysing the protocol unnecessarily, especially when it comes to automating this task. By having the set $\mathcal{P}$ we allow some flexibility in finding a description suitable for automated analysis (the example in Section 4.2.8 illustrates this practice). When every input command in a programme P is specified with $\mathcal{P} = \emptyset$ we say that P is *simple*. Finally, each input command also performs an implicit verification of packages as detailed later; this ensures for instance that a player will only accept packages that were properly created by the other player.

We shall consider four kinds of programmes: *channel*, *plain*, *player*, and *simulator*. The difference between them lies in which kinds of ports they may have and what operations they may use. Figure 4.5 lists all the operations we consider and the following subsections show how these operations are distributed in the systems under consideration. Note here that the available operations implicit limit how they may use the cryptographic material offered to them. Note also that the eval_e method (unlike commit_U and $\mathsf{encrypt}_T$) does not take a randomness

[12] We shall in general omit the "else" part of if-then-else statements if this is just abortion.
[13] Throughout we write operations in this shortened notation, ie. $\mathsf{encrypt}_{bit,R,R}$ instead of $\mathsf{encrypt}_{bit,ek_R,crs_R}$.

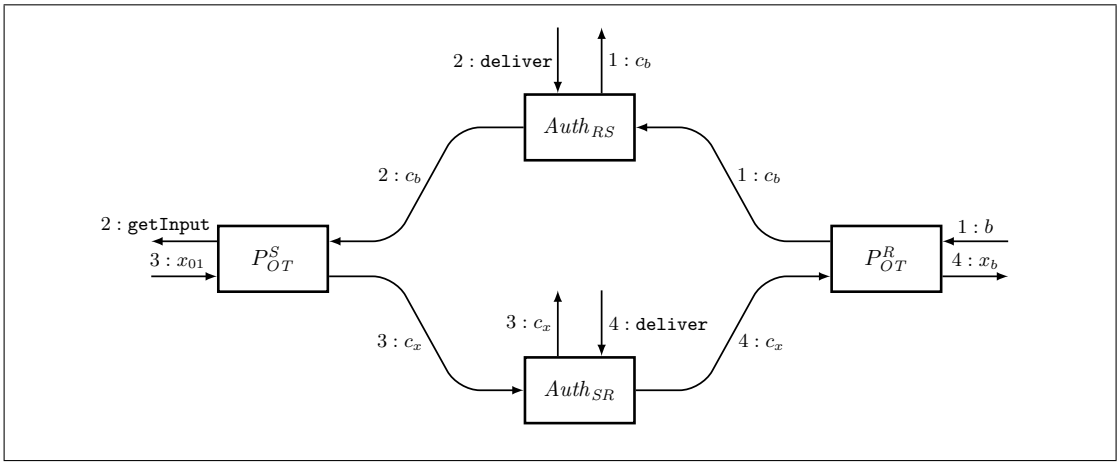

Figure 4.4: Linkes and message flow in the real OT protocol when both players are honest

symbol r as input; we will later come back to this artefact originating from wanting a symbolic model that is easier to analysis with available tools.

The soundness result holds for a larger class of programmes than what can be captured by the simple programming language so to include these we formally define programmes by finite height execution trees. As an example, the OT receiver from above may for instance be expressed as in Figure 4.6.

The nodes Σ are the programme points while the edges specify the actions available at each programme point. These actions come in two flavours, namely *input-output* edges

$$\Sigma \xrightarrow{\text{input } [p_{in}\colon x_{in}] \circ \text{check } \psi \circ \text{compute } \sigma \circ \text{output } [p_{out}\colon x_{out}]} \Sigma'$$

and *input-only* edges

$$\Sigma \xrightarrow{\text{input } [p_{in}\colon x_{in}] \circ \text{check } \psi \circ \text{compute } \sigma} \Sigma'$$

where p_{in}, p_{out} are ports, x_{in}, x_{out} are variables, ψ is a predicate formula over the available testing operations, and $\sigma = \{\frac{\mu_1}{x_1}, \ldots, \frac{\mu_n}{x_n}\}$ is a set of operations μ_i to be executed along with the variables x_i into which the reference to the result is to be stored.

Informally, the execution of an input-output edge happens when there is an accepting input x_{in} on port p_{in} satisfying ψ (whether or not the input is accepting is detailed later). Then the commands in σ are executed as described below and the object pointed to by x_{out} is sent on port p_{out}. Executing an input-only edge is the same except no output is sent. We let Σ be a set containing the references available (defined) at the programme point and constants used to flag specific states.

For a *well-formed* programme we require: (i) that it is deterministic, ie. for any node all the ψ_i on outgoing edges are mutually exclusive per input port yet also together form a tautology; (ii) that it never rebinds a variable; (iii) that it never sends messages directly to itself[14], ie. $\mathbb{P}^{in}_P \cap \mathbb{P}^{out}_P = \emptyset$; (iv) that it never uses a randomness symbol in connection with more than either a commitment or an encryption, ie. that the mapping υ_P from its randomness symbols $\mathbb{R}_P$ to value and kind, $\upsilon_P : \mathbb{R}_P \to \mathbb{V} \times \{\texttt{enc}, \texttt{com}\}$, is a function; and finally, (v) that it only creates each commitment and encryption package once (it may send it several times), ie. it only invokes $\mathsf{commit}_{U,ck,crs}$, $\mathsf{simcommit}_{U,ck,crs}$, $\mathsf{encrypt}_{T,ek,crs}$, or $\mathsf{simencrypt}_{T,ek,crs}$ once per (v, r) pair.

[14]This is because of our encoding in the symbolic model where programmes will not be able to perform a handshake with themself.

$\mathsf{isValue}(x) \to b$ indicates whether x points to a value
$\mathsf{eqValue}(v, w) \to b$ indicates whether v and w point to equal values
$\mathsf{inType}_U(v) \to b$ indicates whether v points to a value in type U
$\mathsf{inType}_T(v) \to b$ indicates whether v points to a value in type T
$\mathsf{peval}_f(v_1, v_2, w_1, w_2) \to v$ evaluates expression e on the values pointed to

$\mathsf{isPair}(x) \to b$ indicates whether x points to a pair
$\mathsf{pair}(x, y) \to z$ creates a pairing of x and y
$\mathsf{first}(z) \to x$ gives a pointer to the first projection of pairing z
$\mathsf{second}(z) \to y$ gives a pointer to the second projection of pairing z

$\mathsf{isConst}(x) \to b$ indicates whether x points to a constant
$\mathsf{eqConst}_c(v) \to b$ indicates whether v points to constant c

$\mathsf{isComPack}(x) \to b$ indicates whether x points to a commitment package
$\mathsf{commit}_{U,ck,crs}(v, r) \to d$ new commitment package for value pointed to by v
$\mathsf{verComPack}_{U,ck,crs}(d) \to b$ indicates whether d is a correct commitment package

$\mathsf{isEncPack}(x) \to b$ indicates whether x points to an encryption package
$\mathsf{encrypt}_{T,ek,crs}(v, r) \to c$ new encryption package for value pointed to by c
$\mathsf{verEncPack}_{T,ek,crs}(c) \to b$ indicates whether c is a correct encryption package

$\mathsf{isEvalPack}(x) \to b$ indicates whether x points to an evaluation package
$\mathsf{eval}_{e,ek,ck,crs}(c_1, c_2, v_1, r_1, v_2, r_2) \to c$ creates evaluation package
$\mathsf{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2) \to b$ indicates whether x is a correct evaluation package
$\mathsf{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2, d_1, d_2) \to b$ indicates whether x is a correct evaluation package

$\mathsf{decrypt}_{dk}(c) \to v$ decrypts encryption pointed to by c

$\mathsf{simcommit}_{U,ck,simtd}(v, r) \to d$ as $\mathsf{commit}_{U,ck,crs}(v, r)$ but ignoring type check
$\mathsf{simencrypt}_{T,ek,simtd}(v, r) \to c$ as $\mathsf{encrypt}_{T,ek,crs}(v, r)$ but ignoring type check
$\mathsf{simeval}_{e,ek,ck,simtd}(c_1, c_2, v_1, r_1, v_2, r_2) \to c$ as $\mathsf{eval}_{e,ek,ck,crs}(c_1, c_2, v_1, r_1, v_2, r_2)$ but simulated proof
$\mathsf{simeval}_{e,ek,ck,simtd}(v, c_1, c_2, d_1, d_2) \to c$ creates fake evaluation package with encryption of v

$\mathsf{extractCom}_{extd}(d) \to v$ extracts value from commitment in com. package
$\mathsf{extractEnc}_{extd}(c) \to v$ extracts value from encryption in encryption package
$\mathsf{extractEval}_{1,extd}(c) \to v$ extracts value from first commitment in eval. package
$\mathsf{extractEval}_{2,extd}(c) \to v$ extracts value from second commitment in eval. package

Figure 4.5: Union of all operations available to the programmes in our protocol class

$$\emptyset \xrightarrow{\text{input } [in^R_{OT}\colon b] \circ \text{check } \mathsf{inType}_{bit}(b) \circ \text{compute } \sigma_1 \circ \text{output } [send_{SR}\colon c_b]} \{b, c_b\}$$

$$\emptyset \xrightarrow{\text{input } [in^R_{OT}\colon b] \circ \text{check } \neg\mathsf{inType}_{bit}(b) \circ \text{compute } \emptyset} \{b, \mathtt{abort}\}$$

$$\{b, c_b\} \xrightarrow{\text{input } [receive_{SR}\colon c_x] \circ \text{check } \mathsf{verEvalPack}_{\ldots}(c_x, c_b) \circ \text{compute } \sigma_2 \circ \text{output } [out^R_{OT}\colon x_b]} \{b, c_b, c_x, x_b\}$$

$$\{b, c_b\} \xrightarrow{\text{input } [receive_{SR}\colon c_x] \circ \text{check } \neg\mathsf{verEvalPack}_{\ldots}(c_x, c_b) \circ \text{compute } \emptyset} \{b, c_b, \mathtt{abort}\}$$

where $\sigma_1 = \left\{ \dfrac{\mathsf{encrypt}_{bit,R,R}(b, r)}{c_b} \right\}$ and $\sigma_2 = \left\{ \dfrac{\mathsf{decrypt}_R(c_x)}{x_b} \right\}$

Figure 4.6: Formal player programme P^R_{OT} for OT receiver

$$Auth_{AB} \doteq \text{input}_{\emptyset}[send_{AB} : x];\ \text{output}[leak_{AB} : x];\ \text{input}_{\emptyset}[infl_{AB} : y];\ \text{output}[receive_{AB} : x];\ \text{stop}$$

Figure 4.7: Programme for an authenticated channel $Auth_{AB}$

$$\mathsf{peval}_e(v_1, v_2, w_1, w_2) \to v$$

$$\begin{array}{rrr} \mathsf{isValue}(x) \to b & \mathsf{isPair}(x) \to b & \\ \mathsf{eqValue}(v, w) \to b & \mathsf{pair}(x, y) \to z & \mathsf{isConst}(x) \to b \\ \mathsf{inType}_U(v) \to b & \mathsf{first}(z) \to x & \mathsf{eqConst}_c(v) \to b \\ \mathsf{inType}_T(v) \to b & \mathsf{second}(z) \to y & \end{array}$$

Figure 4.8: Operations available to plain programmes

Note that condition (v) is only needed to obtain a simplified symbolic model and our models and results may easily be adapted to avoid this condition[15]. Note also that condition (iv) and (v) could be enforced by the operation modules, but to keep these simple we instead add the conditions here.

4.2.4 Real Protocols

The first class of systems in consideration is *real protocols*. The central components of real protocols are given by two player programmes, P^A and P^B. These may have both plain and crypto ports, and may perform cryptographic operations. They may also use *authenticated channels* as a resource; these are simple predefined channel programmes given in Figure 4.7 that accept a single input from one player and delivers it to the another, allowing the adversary to see the transmitted message as well as to choose when it is delivered (but not to change it). The players may also use *ideal functionalities* as a resource; these are triples of plain programmes that may only have plain ports and operate only on values and constants. The programmes have no cryptographic material and may only use the operations in Figure 4.8.

Programme P^A for player A is furthermore given the public key of both parties (ek_A and ek_B), the commitment key of both parties (ck_A and ck_B), the CRS of both parties (crs_A and crs_B), and its own decryption key (dk_A), but it may only use these in accordance with Figure 4.9. The keys and operations given to programme P^B for player B follows symmetrically.

[15]The relevant implication of the condition is that no randomness (or counter) component is needed in the intermediate and symbolic representation of proofs. Our results carry over both if this randomness component is chosen by the adversary or honestly by his operation module (since programmes cannot depend on anything

(as the operations for plain programmes in Figure 4.8...)

$\mathsf{decrypt}_{dk_A}(x) \to v$

$\mathsf{isComPack}(x) \to b$
$\mathsf{commit}_{U,ck_A,crs_A}(v,r) \to d$
$\mathsf{verComPack}_{U,ck_B,crs_B}(d) \to b$

$\mathsf{isEncPack}(x) \to b$
$\mathsf{encrypt}_{T,ek,crs_A}(v,r) \to c$
$\mathsf{verEncPack}_{T,ek,crs_B}(c) \to b$

$\mathsf{isEvalPack}(x) \to b$
$\mathsf{eval}_{e,ek,ck_A,crs_A}(c_1,c_2,v_1,r_1,v_2,r_2) \to c$
$\mathsf{verEvalPack}_{e,ek,ck_B,crs_B}(c,c_1,c_2) \to b$
$\mathsf{verEvalPack}_{e,ek,ck_B,crs_B}(c,c_1,c_2,d_1,d_2) \to b$

Figure 4.9: Operations available to player programme P^A – with $ek \in \{ek_A, ek_B\}$

Denote by $Auth_{AB}$ and $Auth_{BA}$ the parallel composition of two sets of authenticated channels from A to B and B to A, respectively (ie. $Auth_{AB} \doteq Auth_{AB,1} \diamond \cdots \diamond Auth_{AB,n}$). Denote by

$$\mathcal{F}^{AB} \doteq \mathcal{F}_1^{AB} \diamond \cdots \diamond \mathcal{F}_\ell^{AB} \qquad \mathcal{F}^{A} \doteq \mathcal{F}_1^{A} \diamond \cdots \diamond \mathcal{F}_\ell^{A} \qquad \mathcal{F}^{B} \doteq \mathcal{F}_1^{B} \diamond \cdots \diamond \mathcal{F}_\ell^{B}$$

the parallel composition of a set of ℓ functionalities. A real protocol is then defined as follows:

Definition 4.2.1 (Real protocol)**.** *Let the following components be given:*

- *two player programmes P^A and P^B describing the supposed behaviour of A and B*
- *a system $Auth_{AB}$ of authenticated channels from A to B*
- *a system $Auth_{BA}$ of authenticated channels from B to A*
- *a system triple of plain programmes $(\mathcal{F}^{AB}, \mathcal{F}^{A}, \mathcal{F}^{B})$ with the same number in each*

such that the systems

$$Sys_{real}^{AB} \doteq P^A \diamond Auth_{AB} \diamond Auth_{BA} \diamond \mathcal{F}^{AB} \diamond P^B$$
$$Sys_{real}^{A} \doteq P^A \diamond \mathcal{F}^{A}$$
$$Sys_{real}^{B} \doteq P^B \diamond \mathcal{F}^{B}$$

forms a real protocol *through triple $(Sys_{real}^{AB}, Sys_{real}^{A}, Sys_{real}^{B})$ with the player programmes P^A and P^B as the cryptographic programmes.*

Note that the players are not parameterised by the corruption scenario; the same programmes are used in all three cases (but only present if honest). On the other hand, functionalities are allowed to be aware about the corruption scenario.

4.2.5 Ideal Protocols

The other class of systems that we shall consider is *ideal protocols*. The main component of these is a target ideal functionality $\mathcal{F}$ again given by a triple of plain programmes. They also contain simulator programmes that may behave differently depending on the corrupt scenario: in case both players are honest, the simulator programme $Sim^{AB,A}$ for player A may use the operations in Figure 4.10, and symmetrically for the simulator for player B; in case only player A is honest

about proofs except their correctness, in particular not their identity).

(as the operations for plain programmes in Figure 4.8...)

$$\mathsf{isComPack}(x) \to b$$
$$\mathsf{simcommit}_{U,ck_A,simtd_A}(v,r) \to d$$
$$\mathsf{verComPack}_{U,ck_B,crs_B}(d) \to b$$
$$\mathsf{isEncPack}(x) \to b$$
$$\mathsf{simencrypt}_{T,ek,simtd_A}(v,r) \to c$$
$$\mathsf{verEncPack}_{T,ek,crs_B}(c) \to b$$
$$\mathsf{isEvalPack}(x) \to b$$
$$\mathsf{simeval}_{e,ek,ck_A,simtd_A}(c_1,c_2,v_1,r_1,v_2,r_2) \to c$$
$$\mathsf{simeval}_{e,ek,ck_A,simtd_A}(v,c_1,c_2,d_1,d_2) \to c$$
$$\mathsf{verEvalPack}_{e,ek,ck_B,crs_B}(x,x_1,x_2,y_1,y_2) \to b$$

Figure 4.10: Operations available to simulator $Sim^{AB,A}$ when both honest – $ek \in \{ek_A, ek_B\}$

the simulator Sim^A may use the operations in Figure 4.11, and again symmetrically for when only B is honest. Intuitively, the simulators are always offered the public components (ek_A, ek_B, ck_A, ck_B, crs_A and crs_B) and additionally the simulation trapdoor for honest players and the extraction trapdoors for corrupt players. Note that a player programme may be turned into a simulator programme since the latter may use its extraction operations in place of decryption.

Finally, the simulators also have access to the same resources, authenticated channels and functionalities, as a real protocol. However, we here denote the latter as *simulated functionalities* $\mathcal{S}_k$ that are still just triples of plain programmes[16].

(as the operations for simulator programmes in Figure 4.10...)

$$\mathsf{extractEnc}_{extd_B}(c) \to v \qquad \mathsf{extractEval}_{1,extd_B}(c) \to v$$
$$\mathsf{extractCom}_{extd_B}(d) \to v \qquad \mathsf{extractEval}_{2,extd_B}(c) \to v$$

Figure 4.11: Operations available to simulator programme Sim^A when only A is honest

Definition 4.2.2 (Ideal protocol)**.** *Let the following components be given:*

- *a target functionality* $\mathcal{F} = (\mathcal{F}^{AB}, \mathcal{F}^A, \mathcal{F}^B)$
- *two simulator programmes* $Sim^{AB,A}$ *and* $Sim^{AB,B}$ *for when both players are honest*
- *one simulator programme* Sim^A *for when only* A *is honest*
- *one simulator programme* Sim^B *for when only* B *is honest*
- *two systems of authenticated channels* $Auth_{AB}$ *and* $Auth_{BA}$
- *a system triple of plain programmes* $(\mathcal{S}^{AB}, \mathcal{S}^A, \mathcal{S}^B)$ *with the same number in each*

such that the systems

$$Sys^{AB}_{ideal} \doteq \mathcal{F}^{AB} \diamond Sim^{AB,A} \diamond Auth_{AB} \diamond Auth_{BA} \diamond \mathcal{S}^{AB} \diamond Sim^{AB,B}$$
$$Sys^{A}_{ideal} \doteq \mathcal{F}^{A} \diamond Sim^{A} \diamond \mathcal{S}^{A}$$
$$Sys^{B}_{ideal} \doteq \mathcal{F}^{B} \diamond Sim^{B} \diamond \mathcal{S}^{B}$$

forms an ideal protocol Sys_{ideal} *through triple* $(Sys^{AB}_{ideal}, Sys^{A}_{ideal}, Sys^{B}_{ideal})$ *and with the simulators as the cryptographic programmes.*

[16] We use a different name here since the resource functionalities in an ideal protocol need not be related to those found in the real protocol to which the ideal protocol is being compared.

4.2.6 Oblivious Transfer Functionality

As a showcase we here give the complete description of the OT protocol from earlier sections. The real protocol contains the two players programmes P^R_{OT} and P^S_{OT} given in Figure 4.3. Formally the real protocol becomes

$$\left(\quad P^S_{OT} \diamond Auth_{RS} \diamond Auth_{SR} \diamond P^R_{OT} \quad , \quad P^S_{OT} \quad , \quad P^R_{OT} \quad \right)$$

with one authenticated channel in each direction and no functionalities.

For the ideal protocol we first consider the ideal OT functionality and simulators when both players are honest; these are given in Figure 4.12 and we see that the simulators simply run the original protocol on constants. The flow of the protocol for this case is shown in Figure 4.13. For the case where only S is honest we have the ideal functionality and simulator in Figure 4.14 where the simulator extracts the challenge bit b from the encryption sent by the corrupted receiver. For the final case where only R is honest we have the ideal functionality and simulator in Figure 4.15 where the simulator first sends a constant challenge bit zero but then opens the commitments from the corrupt sender to learn both his inputs. The ideal protocol becomes

$$\left(\quad \mathcal{F}^{SR}_{OT} \diamond Sim^{SR,R}_{OT} \diamond Auth_{RS} \diamond Auth_{SR} \diamond Sim^{SR,S}_{OT} \quad , \quad \mathcal{F}^{S}_{OT} \diamond Sim^{S}_{OT} \quad , \quad \mathcal{F}^{R}_{OT} \diamond Sim^{R}_{OT} \quad \right)$$

when combined with the authenticated channels.

In Section 4.7 we use ProVerif to conclude that the systems of these two triples are indistinguishable.

$\mathcal{F}^{SR}_{OT} \doteq \text{input}_{\emptyset}[in^R_{OT} : b];$
if $\mathsf{inType}_{bit}(b)$ then
output$[leak^R_{OT} : \texttt{breceived}];$
input$_{\emptyset}[infl^S_{OT} : \texttt{getInput}];$
output$[out^S_{OT} : \texttt{getInput}];$
input$_{\emptyset}[in^S_{OT} : x_{01}];$
if $\mathsf{isPair}(x_{01})$ then
let $x_0 \leftarrow \mathsf{first}(x_{01});$
let $x_1 \leftarrow \mathsf{second}(x_{01});$
if $\mathsf{isValue}(x_0)$ then
if $\mathsf{isValue}(x_1)$ then
output$[leak^S_{OT} : \texttt{xsreceived}];$
input$_{\emptyset}[infl^R_{OT} : \texttt{finish}];$
if $\mathsf{eqValue}(b, 0)$ then
output$[out^R_{OT} : x_0];$
stop
else
output$[out^R_{OT} : x_1];$
stop

$Sim^{SR,S}_{OT} \doteq \text{input}_{\emptyset}[receive_{RS} : c_b];$
if $\mathsf{verEncPack}_{bit,R,R}(c_b)$ then
output$[infl^S_{OT} : \texttt{getInput}];$
input$_{\emptyset}[leak^S_{OT} : \texttt{xsreceived}];$
let $c_x \leftarrow \mathsf{simeval}_{sel,R,S,S}(c_b, 0, r_0, 0, r_1);$
output$[send_{SR} : c_x];$
stop

$Sim^{SR,R}_{OT} \doteq \text{input}_{\emptyset}[leak^R_{OT} : \texttt{breceived}];$
let $c_b \leftarrow \mathsf{simencrypt}_{bit,R,R}(0, r_b);$
output$[send_{RS} : c_b];$
input$_{\emptyset}[receive_{SR} : c_x];$
if $\mathsf{verEvalPack}_{sel,R,S,S}(c_x, c_b)$ then
output$[infl^R_{OT} : \texttt{finish}];$
stop

Figure 4.12: Ideal OT functionality and simulators for when both players are honest

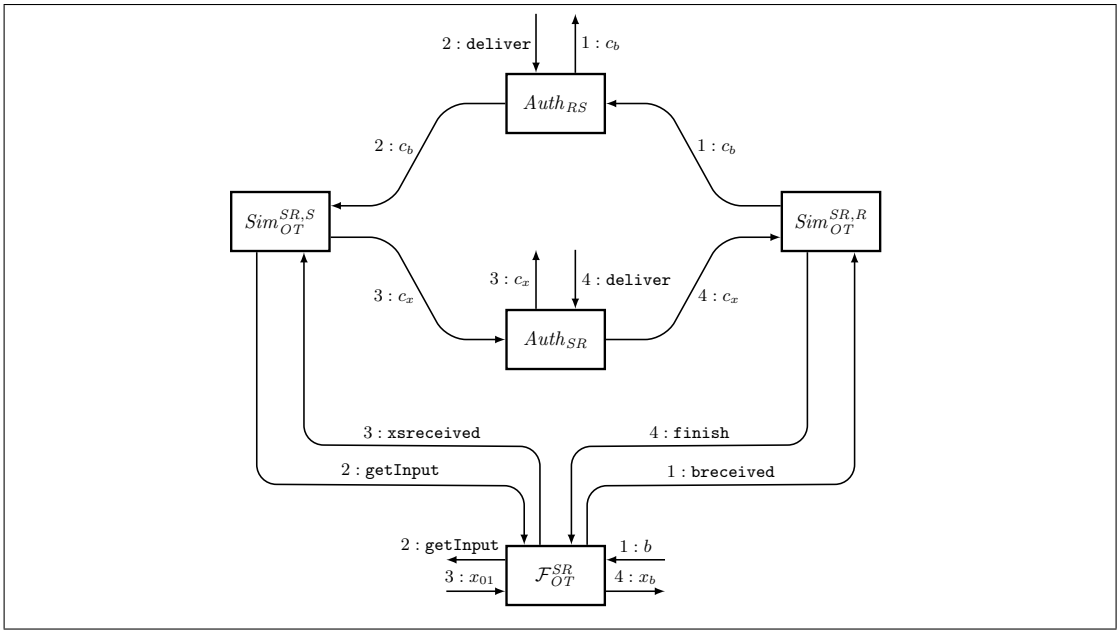

Figure 4.13: Links and message flow of ideal OT protocol when both players are honest

$\mathcal{F}_{OT}^{S} \doteq \text{input}_{\emptyset}[\mathit{infl}_{OT} : b];$
if $\mathsf{inType}_{bit}(b)$ then
output$[\mathit{out}_{OT}^{S} : \texttt{getInput}]$;
input$_{\emptyset}[\mathit{in}_{OT}^{S} : x_{01}]$;
if $\mathsf{isPair}(x_{01})$ then
let $x_0 \leftarrow \mathsf{first}(x_{01})$;
let $x_1 \leftarrow \mathsf{second}(x_{01})$;
if $\mathsf{isValue}(x_0)$ then
if $\mathsf{isValue}(x_1)$ then
if $\mathsf{eqValue}(b, 0)$ then
output$[\mathit{leak}_{OT} : x_0]$;
stop
else
output$[\mathit{leak}_{OT} : x_1]$;
stop

$\mathit{Sim}_{OT}^{S} \doteq \text{input}_{\emptyset}[\mathit{receive}_{RS} : c_b];$
if $\mathsf{verEncPack}_{bit,R,R}(c_b)$ then
let $b \leftarrow \mathsf{extractEnc}_R(c_b)$;
output$[\mathit{infl}_{OT} : b]$;
input$_{\emptyset}[\mathit{leak}_{OT} : x_b]$;
let $c_x \leftarrow \mathsf{simeval}_{sel,R,S,S}(x_b, c_b, 0, r_0, 0, r_1)$;
output$[\mathit{send}_{SR} : c_x]$;
stop

Figure 4.14: Ideal OT functionality and simulator for when only S is honest

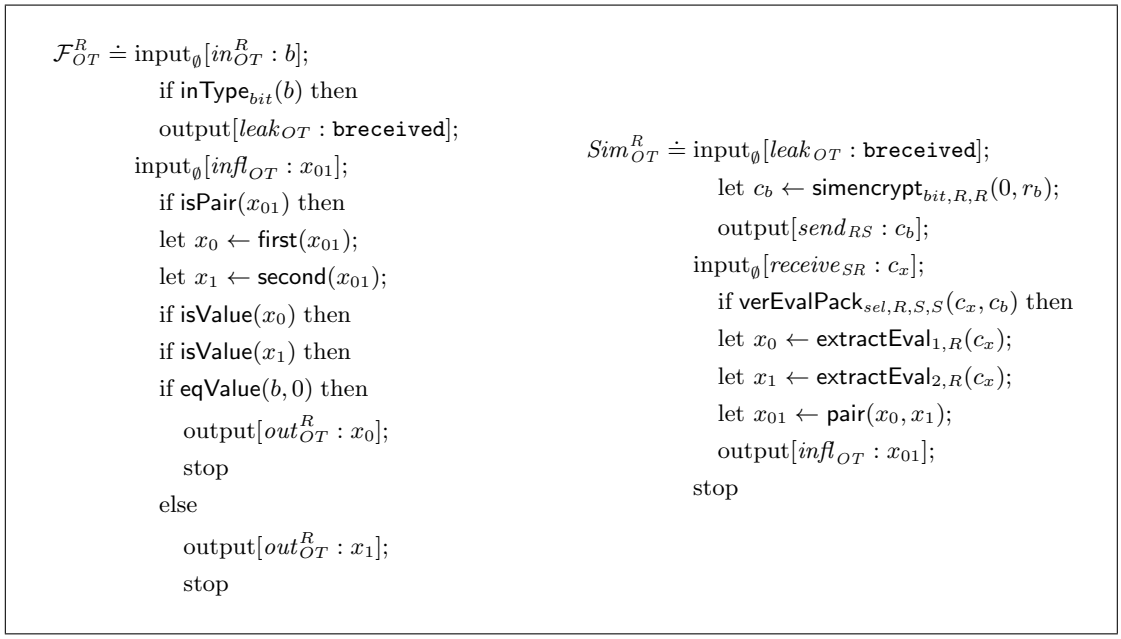

Figure 4.15: Ideal OT functionality and simulator for when only R is honest

4.2.7 Commitment Functionality

As a stepping stone towards presenting the coin-flip (CF) functionality below, we here give a generic commitment functionality parameterised by a type *dom* for the committed value. Note that in the CF functionality below we only use the ideal commitment functionality $\mathcal{F}_{com}$ presented here and hence together these examples also illustrates compositional analysis of protocols.

The commitment functionality intuitively allows a *committer* C to convince *opener* O that he has committed himself to a value v, without revealing v to O. At a later point C may choose to open the commitment and reveal the value to O, at the same time convincing him that the value he learns is really the value v committed to earlier.

Figure 4.16 shows player programme P^C for *committer* C and P^O for *opener* O. The expression is given by $minus \doteq \alpha - \beta$ and is used to check that the value in commitment d send by C in the first step is also the value in encryption c send in the final step when opening with (c, c_0); the check performed by O verifies that c_0 decrypts to 0. Note that the `ack` step is included so that we may again only use simple programmes[17]. The real protocol becomes

$$\left(\quad P^C_{com} \diamond Auth_{CO,1} \diamond Auth_{OC} \diamond Auth_{CO,2} \diamond P^O_{com} \quad , \quad P^C_{com} \quad , \quad P^O_{com} \quad \right)$$

with three authenticated channels and no resource functionalities.

The ideal commitment functionality $\mathcal{F}_{com}$ and simulators for the three corruption scenarios are given in Figure 4.17, 4.18, and Figure 4.19. The ideal protocol becomes

$$\left(\quad \mathcal{F}^{CO}_{com} \diamond Sim^{CO,C}_{com} \diamond Auth_{CO,1} \diamond Auth_{OC} \diamond Auth_{CO,2} \diamond Sim^{CO,O}_{com} \quad , \right.$$
$$\left. \mathcal{F}^{C}_{com} \diamond Sim^{S}_{com} \quad , \quad \mathcal{F}^{O}_{com} \diamond Sim^{R}_{com} \quad \right)$$

again with three authenticated channels and no resource functionalities.

$P^C_{com} \doteq$
- $\text{input}_{\emptyset}[in^C_{com} : x]$;
- if $\mathsf{inType}_{dom}(x)$ then
- let $d \leftarrow \mathsf{commit}_{dom,C,C}(x, r_d)$;
- $\text{output}[send_{CO,1} : d]$;
- $\text{input}_{\emptyset}[receive_{OC} : \texttt{ack}]$;
- $\text{output}[out^C_{com} : \texttt{ack}]$;
- $\text{input}_{\emptyset}[in^C_{com} : \texttt{open}]$;
- let $c \leftarrow \mathsf{encrypt}_{dom,O,C}(x, r_c)$;
- let $c_0 \leftarrow \mathsf{eval}_{minus,O,C,C}(c, x, r_d)$;
- $\text{output}[send_{CO,2} : (c, c_0)]$;
- stop

$P^O_{com} \doteq$
- $\text{input}_{\emptyset}[receive_{CO,1} : d]$;
- if $\mathsf{verComPack}_{dom,C,C}(d)$ then
- $\text{output}[out^O_{com} : \texttt{committed}]$;
- $\text{input}_{\emptyset}[in^O_{com} : \texttt{ack}]$;
- $\text{output}[send_{OC} : \texttt{ack}]$;
- $\text{input}_{\emptyset}[receive_{CO,2} : (c, c_0)]$;
- if $\mathsf{verEncPack}_{dom,O,C}(c)$ then
- if $\mathsf{verEvalPack}_{minus,O,C,C}(c_0, c, d)$ then
- let $x_0 \leftarrow \mathsf{decrypt}_O(c_0)$;
- if $\mathsf{eqValue}(x_0, 0)$ then
- let $x \leftarrow \mathsf{decrypt}_O(c)$;
- $\text{output}[out^O_{com} : x]$;
- stop

Figure 4.16: Player programme P^C_{com} for committer (left) and P^O_{com} for opener (right)

[17] Without the `ack` message the adversary may force the opener to receive the opening (c, c_0) before the commitment d; to ensure that no message is lost we could then no longer describe the opener by a simple programme.

$\mathcal{F}_{com}^{CO} \doteq \text{input}_{\emptyset}[in_{com}^{C} : x];$
if $\mathsf{inType}_{dom}(x)$ then
output$[leak_{com}^{C} : \texttt{delayedcommitted}];$
$\text{input}_{\emptyset}[infl_{com}^{O} : \texttt{continue}];$
output$[out_{com}^{O} : \texttt{committed}];$
$\text{input}_{\emptyset}[in_{com}^{O} : \texttt{ack}];$
output$[leak_{com}^{O} : \texttt{delayedack}];$
$\text{input}_{\emptyset}[infl_{com}^{C} : \texttt{continue}];$
output$[out_{com}^{C} : \texttt{ack}]$
$\text{input}_{\emptyset}[in_{com}^{C} : \texttt{open}];$
output$[leak_{com}^{C} : \texttt{delayedopen}];$
$\text{input}_{\emptyset}[infl_{com}^{O} : \texttt{continue}];$
output$[out_{com}^{O} : x];$
stop

$Sim_{com}^{CO,C} \doteq \text{input}_{\emptyset}[leak_{com}^{C} : \texttt{delayedcommitted}];$
let $d \leftarrow \mathsf{simcommit}_{dom,C,C}(0, r_d);$
output$[send_{CO,1} : d];$
$\text{input}_{\emptyset}[receive_{OC} : \texttt{ack}];$
output$[infl_{com}^{C} : \texttt{continue}];$
$\text{input}_{\emptyset}[leak_{com}^{C} : \texttt{delayedopen}];$
let $c \leftarrow \mathsf{simencrypt}_{dom,O,C}(0, r_c);$
let $c_0 \leftarrow \mathsf{simeval}_{minus,O,C,C}(c, 0, r_d);$
output$[send_{CO,2} : (c, c_0)];$
stop

$Sim_{com}^{CO,C} \doteq \text{input}_{\emptyset}[receive_{CO,1} : d];$
output$[infl_{com}^{O} : \texttt{continue}];$
$\text{input}_{\emptyset}[leak_{com}^{O} : \texttt{delayedack}];$
output$[send_{OC} : \texttt{ack}];$
$\text{input}_{\emptyset}[receive_{CO,2} : (c, c_0)];$
output$[infl_{com}^{O} : \texttt{continue}];$
stop

Figure 4.17: Ideal commitment functionality and simulators for when both players are honest

$\mathcal{F}_{com}^{C} \doteq \text{input}_{\emptyset}[in_{com}^{C} : x];$
if $\mathsf{inType}_{dom}(x)$ then
output$[leak_{com} : \texttt{committed}];$
$\text{input}_{\emptyset}[infl_{com} : \texttt{ack}];$
output$[out_{com}^{C} : \texttt{ack}]$
$\text{input}_{\emptyset}[in_{com}^{C} : \texttt{open}];$
output$[leak_{com} : x];$
stop

$Sim_{com}^{C} \doteq \text{input}_{\emptyset}[leak_{com} : \texttt{committed}];$
let $d \leftarrow \mathsf{simcommit}_{dom,C,C}(0, r_d);$
output$[send_{CO,1} : d];$
$\text{input}_{\emptyset}[receive_{OC} : \texttt{ack}];$
output$[infl_{com} : \texttt{ack}];$
$\text{input}_{\emptyset}[leak_{com} : x];$
let $c \leftarrow \mathsf{simencrypt}_{dom,O,C}(x, r_c);$
let $c_0 \leftarrow \mathsf{simeval}_{minus,O,C,C}(0, c, d);$
output$[send_{CO,2} : (c, c_0)];$
stop

Figure 4.18: Ideal commitment functionality and simulator for when only committer is honest

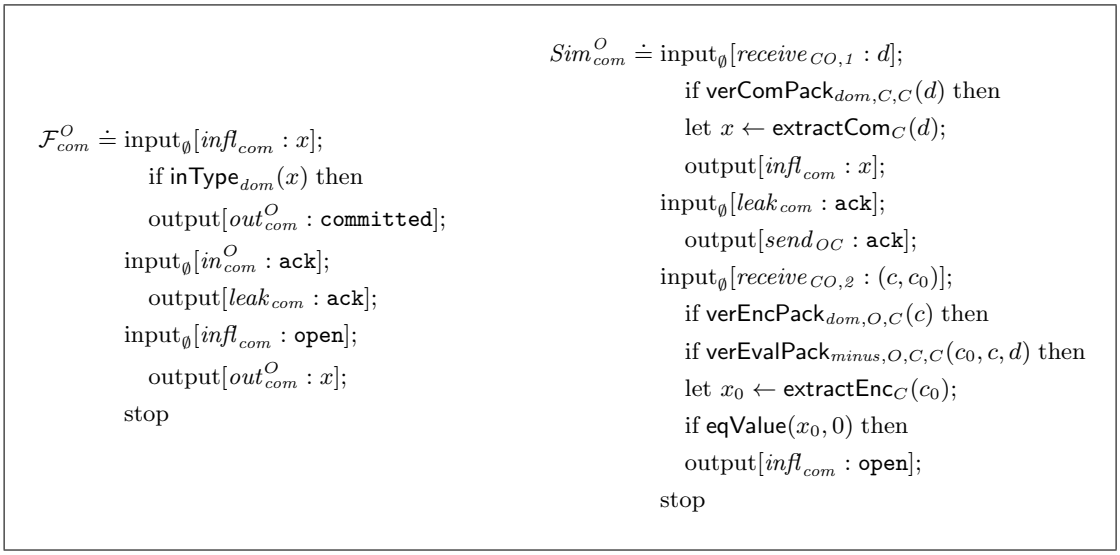

Figure 4.19: Ideal commitment functionality and simulator for when only opener is honest

4.2.8 Coin-flip Functionality

Our coin-flip functionality takes bit a from A and bit b from B as input, and returns $c = a \oplus b$ to both of them[18]. The security guarantee is that both coins are chosen independently. This is ensured by first letting A commit to a using commitment functionality $\mathcal{F}_{com}$ from above. Then B sends b to A in cleartext so that she may compute c. Finally, A opens her commitment to B who may now also compute c.

Let $\mathcal{F}_{com}$ be the commitment functionality from above instantiated with $dom = bit$, and define expression $xor(\alpha_1, \alpha_2) \doteq \alpha_1 + \alpha_2 - 2 \cdot \alpha_1 \cdot \alpha_2$. The programme P^A_{CF} for player A is then given in Figure 4.20 together with programme P^B_{CF} for player B. Note that since the protocol uses the commitment functionality, P^B_{CF} must use $\text{input}_{\{out^O_{com}\}}[\cdot : \cdot]$ twice to ensure that no message is lost in the scenario where A is corrupt and the adversary instructs the commitment functionality to open before it is supposed to. When both players are honest they may be removed to yield a simple programme. The real protocol becomes

$$\Big(\quad P^A_{CF} \diamond Auth_{AB} \diamond Auth_{BA} \diamond \mathcal{F}^{CO}_{com} \diamond P^B_{CF} \quad , \quad P^A_{CF} \diamond \mathcal{F}^C_{com} \quad , \quad P^B_{CF} \diamond \mathcal{F}^O_{com} \quad \Big)$$

with two authenticated channels and the commitment functionality.

The ideal coin-flip functionality $\mathcal{F}_{CF}$, its simulators, and simulated commitment functionality $\mathcal{S}_{com}$ are given in Figure 4.21, 4.22, and 4.23 for the three corruption scenarios. The simulated commitment functionality differs from $\mathcal{F}_{com}$ in that the committer specifies the "committed" value when opening. The ideal protocol becomes

$$\Big(\quad \mathcal{F}^{AB}_{CF} \diamond Sim^{AB,A}_{CF} \diamond Auth_{AB} \diamond Auth_{BA} \diamond \mathcal{S}^{CO}_{com} \diamond Sim^{AB,B}_{CF} \quad , \quad \mathcal{F}^A_{CF} \diamond Sim^A_{CF} \diamond \mathcal{S}^C_{com} \quad , \quad \mathcal{F}^B_{CF} \diamond Sim^B_{CF} \diamond \mathcal{S}^O_{com} \quad \Big)$$

with two authenticated channels and the simulated commitment functionality.

$P^A_{CF} \doteq \text{input}_{\emptyset}[in^A_{cf} : a];$
if $\mathsf{inType}_{bit}(a)$ then
output$[in^C_{com} : d];$
input$_{\emptyset}[out^C_{com} : \texttt{ack}];$
output$[send_{AB} : \texttt{proceed}];$
input$_{\emptyset}[receive_{BA} : b];$
if $\mathsf{inType}_{bit}(b)$ then
let $c \leftarrow \mathsf{peval}_{xor}(a, b);$
output$[out^A_{cf} : c];$
input$_{\emptyset}[in^A_{cf} : \texttt{outputToB}];$
output$[in^C_{com} : \texttt{open}];$
stop

$P^B_{CF} \doteq \text{input}_{\emptyset}[out^O_{com} : \texttt{committed}];$
output$[in^O_{com} : \texttt{ack}];$
input$_{\{out^O_{com}\}}[receive_{AB} : \texttt{proceed}];$
output$[out^B_{cf} : \texttt{getInput}];$
input$_{\{out^O_{com}\}}[in^B_{cf} : b];$
if $\mathsf{inType}_{bit}(b)$ then
output$[send_{BA} : b];$
input$_{\emptyset}[out^O_{com} : a];$
let $c \leftarrow \mathsf{peval}_{xor}(a, b);$
output$[out^B_{cf} : c];$
stop

Figure 4.20: Player programme P^A_{CF} for A (left) and P^B_{CF} for B (right)

[18] Note that our protocol model does not allow programmes (and hence players) to pick a random value. In the case of coin-flipping this means that the coins must come from the environment. This also implies that functionalities cannot abstract output distributions as is typically done. Further discussion is given in Section 4.8.

$Sim_{CF}^{AB,A} \doteq \text{input}_{\emptyset}[leak_{cf}^{A} : \texttt{receivedInputFromA}];$
$\quad \text{output}[in_{com}^{C} : \texttt{fakeCommit}];$
$\quad \text{input}_{\emptyset}[out_{com}^{C} : \texttt{ack}];$
$\quad \text{output}[send_{AB} : \texttt{proceed}];$
$\quad \text{input}_{\emptyset}[receive_{BA} : b];$
$\quad \text{output}[infl_{cf}^{A} : \texttt{outputToA}];$
$\quad \text{input}_{\emptyset}[leak_{cf}^{A} : a];$
$\quad \text{output}[in_{com}^{C} : a];$
$\quad \text{stop}$

$Sim_{CF}^{AB,B} \doteq \text{input}_{\emptyset}[out_{com}^{O} : \texttt{committed}];$
$\quad \text{output}[in_{com}^{O} : \texttt{ack}];$
$\quad \text{input}_{\emptyset}[receive_{AB} : \texttt{proceed}];$
$\quad \text{output}[infl_{cf}^{B} : \texttt{getInputFromB}];$
$\quad \text{input}_{\emptyset}[leak_{cf}^{B} : b];$
$\quad \text{output}[send_{BA} : b];$
$\quad \text{input}_{\emptyset}[out_{com}^{O} : a];$
$\quad \text{output}[infl_{cf}^{B} : \texttt{continue}];$
$\quad \text{stop}$

$\mathcal{F}_{CF}^{AB} \doteq \text{input}_{\emptyset}[in_{cf}^{A} : a];$
$\quad \text{if } \mathsf{inType}_{bit}(a) \text{ then}$
$\quad \text{output}[leak_{cf}^{A} : \texttt{receivedInputFromA}];$
$\quad \text{input}_{\emptyset}[infl_{cf}^{B} : \texttt{getInputFromB}];$
$\quad \text{output}[out_{cf}^{B} : \texttt{getInput}];$
$\quad \text{input}_{\emptyset}[in_{cf}^{B} : b];$
$\quad \text{if } \mathsf{inType}_{bit}(b) \text{ then}$
$\quad \text{output}[leak_{cf}^{B} : b];$
$\quad \text{input}_{\emptyset}[infl_{cf}^{A} : \texttt{outputToA}];$
$\quad \text{let } c \leftarrow \mathsf{peval}_{xor}(a, b);$
$\quad \text{output}[out_{cf}^{A} : c]$
$\quad \text{input}_{\emptyset}[in_{cf}^{A} : \texttt{outputToB}];$
$\quad \text{output}[leak_{cf}^{A} : a];$
$\quad \text{input}_{\emptyset}[infl_{cf}^{B} : \texttt{continue}];$
$\quad \text{output}[out_{cf}^{B} : c];$
$\quad \text{stop}$

$\mathcal{S}_{com}^{CO} \doteq \text{input}_{\emptyset}[in_{com}^{C} : \texttt{fakeCommit}];$
$\quad \text{output}[leak_{com}^{C} : \texttt{delayedcommitted}];$
$\quad \text{input}_{\emptyset}[infl_{com}^{O} : \texttt{continue}];$
$\quad \text{output}[out_{com}^{O} : \texttt{committed}];$
$\quad \text{input}_{\emptyset}[in_{com}^{O} : \texttt{ack}];$
$\quad \text{output}[leak_{com}^{O} : \texttt{delayedack}];$
$\quad \text{input}_{\emptyset}[infl_{com}^{C} : \texttt{continue}];$
$\quad \text{output}[out_{com}^{C} : \texttt{ack}]$
$\quad \text{input}_{\emptyset}[in_{com}^{C} : x];$
$\quad \text{output}[leak_{com}^{C} : \texttt{delayedopen}];$
$\quad \text{input}_{\emptyset}[infl_{com}^{O} : \texttt{continue}];$
$\quad \text{output}[out_{com}^{O} : x];$
$\quad \text{stop}$

Figure 4.21: Ideal CF functionality, simulators, and simulated functionality when both are honest

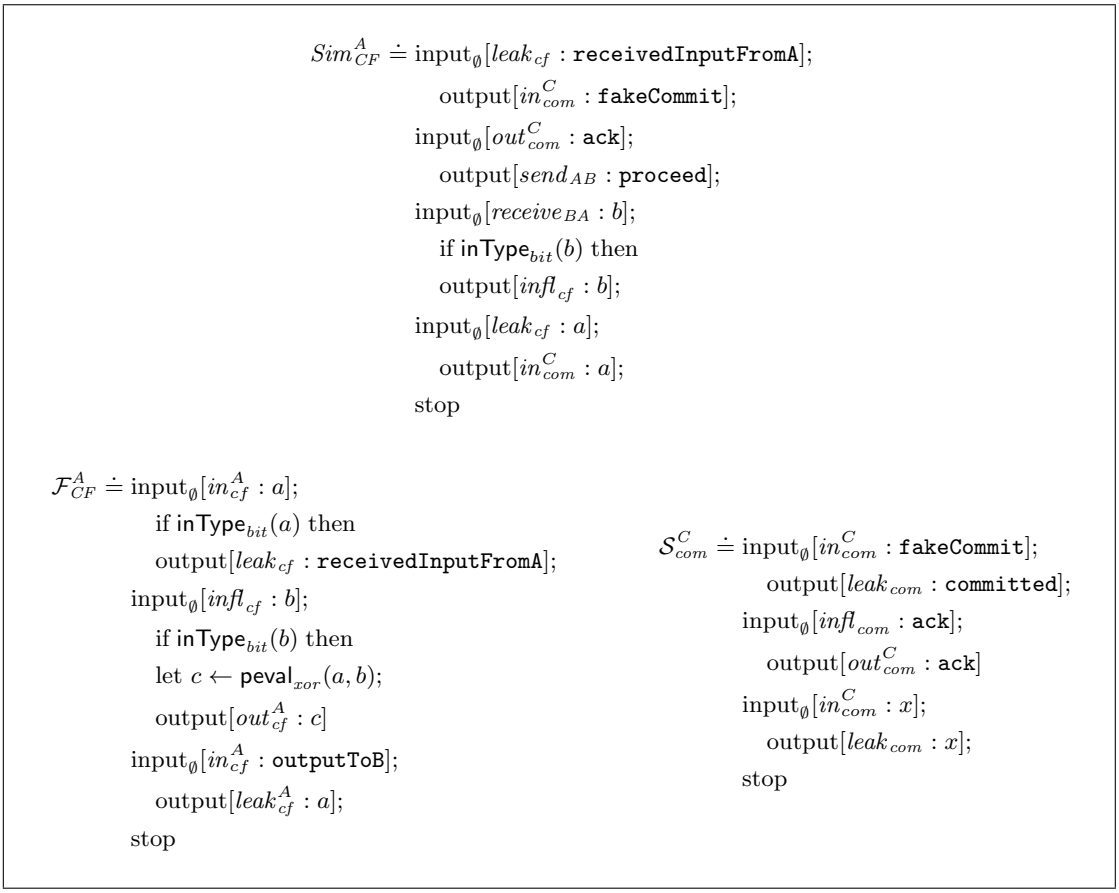

Figure 4.22: Ideal CF functionality, simulator, and simulated functionality for only A honest

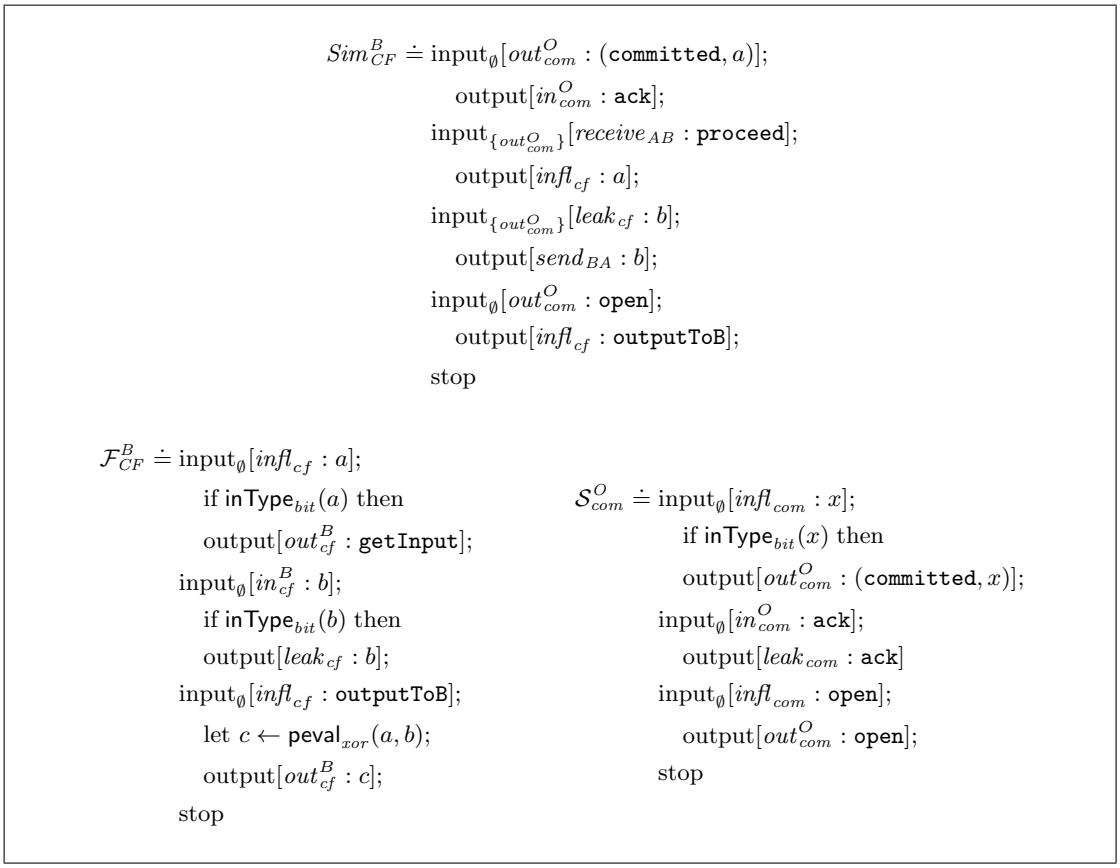

Figure 4.23: Ideal CF functionality, simulator, and simulated functionality when only B honest

4.2.9 Multiplication Triple Functionality

As an exercise in expressibility[19] we next consider the Π_{trip} protocol given by Bendlin et al. in [BDOZ11], and used in the offline phase of their MPC protocol to securely generate random shares of multiplication triples between a set of players. We instantiate the protocol for the generation of a single triple between two players, denoted 1 and 2, under static corruption. In other words, the two players respectively generate shares (a_1, b_1, c_1) and (a_2, b_2, c_2) with information theoretic MACs. Moreover, since we cannot model the probabilistic choice[20] in the final check of Π_{trip} we consider a variant where this check is pushed to the online phase (much like in [DPSZ12]) and allowing an error: define $a = a_1 + a_2$, $b = b_1 + b_2$, and $c = c_1 + c_2$; when both players are honest our variation makes no difference and we require that $a \cdot b = c$ as in the original protocol; however, when one player is corrupt we now require that $a \cdot b = c + e$ for some error e known by the adversary.

The two player programmes P^1_{trip} and P^2_{trip} are given in Figure 4.24 and 4.25 respectively; since the protocol is already complex enough, we here present it slightly informally for readability, yet expressing all programmes in our formal language is straight-forward. The first thing to notice it that the two players receive all their random choices from the environment. This is again because we cannot model probabilistic choice, yet if this formulation is secure then clearly the formulation where the randomness is instead drawn honestly from a distribution is also secure. One consequence of this is that the same exact values must now be computed by the protocol and the ideal functionality, and hence the latter is now slightly less abstract than what we might prefer as we shall see. The second thing to notice is that the players are now sending an encryption of 0 together with the initial commitment to their α. This is because we cannot directly give a proof that a ciphertext under encryption ek was constructed using the plaintext value of another ciphertext under a different encryption key. Concretely, we for instance have that when player 1 commits to a_1 by sending ciphertext[21] $\mathcal{C}_{a_1}$ he must do so under his own encryption key ek_1 to keep it secret; however he must also prove that he used the same plaintext when he later constructs $\mathcal{C}_{x_2}$ under ek_2.

The ideal functionality $\mathcal{F}^{12}_{trip}$ for when both players are honest are given in Figure 4.26. The only thing to notice here is that c_1 is now computed to match the exact value returned by player 1. However, in the formulation where z_1 is honestly drawn from a distribution instead of being provided by the environment c_1 is effectively drawn as in the original ideal functionality. Simulators for when both players are honest is given in Figure 4.27 and 4.28; they simply execute the protocol using constants.

Figure 4.29 gives the ideal functionality $\mathcal{F}^1_{trip}$ when only player 1 is honest. Unlike $\mathcal{F}^{12}_{trip}$ it now expects the adversary to provide an error value e such that $c = c_1 + c_2 = e$; the simulator in Figure 4.30 computes this value by extracting values from the corrupted player 2.

Finally, the ideal functionality $\mathcal{F}^1_{trip}$ for the symmetric case where only player 2 is honest is given in Figure 4.31 and its simulator in Figure 4.32. Here the error value e is extracted from player 1 instead.

[19] Our attempts at verifying the equivalences using ProVerif were not conclusive as the tool never terminated. A significant factor here seemed to be the many input parameters needed to model the probabilistic choices.

[20] While we may analyse some probabilistic choices, the kind used here falls into another category which it is unclear how to capture; see Section 4.8 for more discussion.

[21] Note that he cannot commit to a_1 using only a commitment since player 2 later needs $\mathcal{C}_{a_1}$ to form $\mathcal{C}_{m(a_1)}$.

1. On input $(\alpha_2, a_1, b_1, \beta_{a_2}, \beta_{b_2}, \beta_{c_2}, z_1)$ on port in^1_{trip}
   a) check that all values are in the domain
   b) let $\mathcal{D}_{\alpha_2} \leftarrow \mathsf{commit}_{dom,ck_1,crs_1}(\alpha_2, r_{\alpha_2})$
   c) let $\mathcal{C}_{0_2} \leftarrow \mathsf{encrypt}_{dom,ek_2,crs_1}(0, r_{0_2})$
   d) send $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ to player 2
2. On receiving $(\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1})$ from player 2
   a) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{\alpha_1})$ and $\mathsf{verEncPack}_{dom,ek_1,crs_2}(\mathcal{C}_{0_1})$
   b) check $\mathsf{decrypt}_{dk_1}(\mathcal{C}_{0_1}) = 0$
   c) let $\mathcal{D}_{a_1} \leftarrow \mathsf{commit}_{dom,ck_1,crs_1}(a_1, r_{a_1})$ and $\mathcal{C}_{a_1} \leftarrow \mathsf{eval}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{0_1}, a_1, r_{a_1})$
   d) let $\mathcal{D}_{b_1} \leftarrow \mathsf{commit}_{dom,ck_1,crs_1}(b_1, r_{b_1})$ and $\mathcal{C}_{b_1} \leftarrow \mathsf{eval}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{0_1}, b_1, r_{b_1})$
   e) send $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ to player 2
3. On receiving $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ from player 2
   a) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{a_2})$ and $\mathsf{verEvalPack}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{a_2}, \mathcal{C}_{0_2}, \mathcal{D}_{a_2})$
   b) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{b_2})$ and $\mathsf{verEvalPack}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{b_2}, \mathcal{C}_{0_2}, \mathcal{D}_{b_2})$
   c) let $\mathcal{C}_{x_2} \leftarrow \mathsf{eval}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{b_2}, a_1, r_{a_1}, z_1, r_{z_1})$
   d) send $\mathcal{C}_{x_2}$ to player 2
4. On receiving $\mathcal{C}_{x_1}$ from player 2
   a) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{x_1}, \mathcal{C}_{b_1}, \mathcal{D}_{a_2})$
   b) let $x_1 \leftarrow \mathsf{decrypt}_{dk_1}(\mathcal{C}_{x_1})$
   c) let $c_1 = a_1 \cdot b_1 + x_1 - z_1$
   d) let $\mathcal{C}_{c_1} \leftarrow \mathsf{encrypt}_{dom,ek_1,crs_1}(c_1, r_{c_1})$
   e) send $\mathcal{C}_{c_1}$ to player 2
5. On receiving $\mathcal{C}_{c_2}$ from player 2
   a) check $\mathsf{verEncPack}_{dom,ek_2,crs_2}(\mathcal{C}_{c_2})$
   b) let $\mathcal{C}_{m(a_2)} \leftarrow \mathsf{eval}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{a_2}, \alpha_2, r_{\alpha_2}, \beta_{a_2}, r_{\beta_{a_2}})$
   c) let $\mathcal{C}_{m(b_2)} \leftarrow \mathsf{eval}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{b_2}, \alpha_2, r_{\alpha_2}, \beta_{b_2}, r_{\beta_{b_2}})$
   d) let $\mathcal{C}_{m(c_2)} \leftarrow \mathsf{eval}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{c_2}, \alpha_2, r_{\alpha_2}, \beta_{c_2}, r_{\beta_{c_2}})$
   e) send $\left(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\right)$ to player 2
6. On receiving $\left(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\right)$ from player 2
   a) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(a_1)}, \mathcal{C}_{a_1}, \mathcal{D}_{\alpha_1})$
   b) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(b_1)}, \mathcal{C}_{b_1}, \mathcal{D}_{\alpha_1})$
   c) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(c_1)}, \mathcal{C}_{c_1}, \mathcal{D}_{\alpha_1})$
   d) let $m(a_1) \leftarrow \mathsf{decrypt}_{dk_1}(\mathcal{C}_{m(a_1)})$
   e) let $m(b_1) \leftarrow \mathsf{decrypt}_{dk_1}(\mathcal{C}_{m(b_1)})$
   f) let $m(c_1) \leftarrow \mathsf{decrypt}_{dk_1}(\mathcal{C}_{m(c_1)})$
   g) output $\left(c_1, m(a_1), m(b_1), m(c_1)\right)$ on port out^1_{trip}

Figure 4.24: Player P^1_{trip} for multiplication triple generation

1. On receiving $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ from player 1
   a) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{\alpha_2})$ and $\mathsf{verEncPack}_{dom,ek_2,crs_1}(\mathcal{C}_{0_2})$
   b) check $\mathsf{decrypt}_{dk_2}(\mathcal{C}_{0_2}) = 0$
   c) output `getInput` on port out^2_{trip}
2. On input $(\alpha_1, a_2, b_2, \beta_{a_1}, \beta_{b_1}, \beta_{c_1}, z_2)$ on port in^2_{trip}
   a) check that all values are in the domain
   b) let $\mathcal{D}_{\alpha_1} \leftarrow \mathsf{commit}_{dom,ck_2,crs_2}(\alpha_1, r_{\alpha_1})$
   c) let $\mathcal{C}_{0_1} \leftarrow \mathsf{encrypt}_{dom,ek_1,crs_2}(0, r_{0_1})$
   d) send $(\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1})$ to player 1
3. On receiving $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ from player 1
   a) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{a_1})$ and $\mathsf{verEvalPack}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{a_1}, \mathcal{C}_{0_1}, \mathcal{D}_{a_1})$
   b) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{b_1})$ and $\mathsf{verEvalPack}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{b_1}, \mathcal{C}_{0_1}, \mathcal{D}_{b_1})$
   c) let $\mathcal{D}_{a_2} \leftarrow \mathsf{commit}_{dom,ck_2,crs_2}(a_2, r_{a_2})$ and $\mathcal{C}_{a_2} \leftarrow \mathsf{eval}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{0_2}, a_2, r_{a_2})$
   d) let $\mathcal{D}_{b_2} \leftarrow \mathsf{commit}_{dom,ck_2,crs_2}(b_2, r_{b_2})$ and $\mathcal{C}_{b_2} \leftarrow \mathsf{eval}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{0_2}, b_2, r_{b_2})$
   e) send $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ to player 1
4. On receiving $\mathcal{C}_{x_2}$ from player 1
   a) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{x_2}, \mathcal{C}_{b_2}, \mathcal{D}_{a_1})$
   b) let $x_2 \leftarrow \mathsf{decrypt}_{dk_2}(\mathcal{C}_{x_2})$
   c) let $\mathcal{C}_{x_1} \leftarrow \mathsf{eval}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{b_1}, a_2, r_{a_2}, z_2, r_{z_2})$
   d) send $\mathcal{C}_{x_1}$ to player 1
5. On receiving $\mathcal{C}_{c_1}$ from player 1
   a) check $\mathsf{verEncPack}_{dom,ek_1,crs_1}(\mathcal{C}_{c_1})$
   b) let $c_2 = a_2 \cdot b_2 + x_2 - z_2$
   c) let $\mathcal{C}_{c_2} \leftarrow \mathsf{encrypt}_{dom,ek_2,crs_2}(c_2, r_{c_2})$
   d) send $\mathcal{C}_{c_2}$ to player 1
6. On receiving $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ from player 1
   a) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(a_2)}, \mathcal{C}_{a_2}, \mathcal{D}_{\alpha_2})$
   b) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(b_2)}, \mathcal{C}_{b_2}, \mathcal{D}_{\alpha_2})$
   c) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(c_2)}, \mathcal{C}_{c_2}, \mathcal{D}_{\alpha_2})$
   d) let $m(a_2) \leftarrow \mathsf{decrypt}_{dk_2}(\mathcal{C}_{m(a_2)})$
   e) let $m(b_2) \leftarrow \mathsf{decrypt}_{dk_2}(\mathcal{C}_{m(b_2)})$
   f) let $m(c_2) \leftarrow \mathsf{decrypt}_{dk_2}(\mathcal{C}_{m(c_2)})$
   g) output $\big(c_2, m(a_2), m(b_2), m(c_2)\big)$ on port out^2_{trip}
7. On input `outputTo1` on port in^2_{trip}
   a) let $\mathcal{C}_{m(a_1)} \leftarrow \mathsf{eval}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{a_1}, \alpha_1, r_{\alpha_1}, \beta_{a_1}, r_{\beta_{a_1}})$
   b) let $\mathcal{C}_{m(b_1)} \leftarrow \mathsf{eval}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{b_1}, \alpha_1, r_{\alpha_1}, \beta_{b_1}, r_{\beta_{b_1}})$
   c) let $\mathcal{C}_{m(c_1)} \leftarrow \mathsf{eval}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{c_1}, \alpha_1, r_{\alpha_1}, \beta_{c_1}, r_{\beta_{c_1}})$
   d) send $\big(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\big)$ to player 1

Figure 4.25: Player P^2_{trip} for multiplication triple generation

1. On input $(\alpha_2, a_1, b_1, \beta_{a_2}, \beta_{b_2}, \beta_{c_2}, z_1)$ on port in^1_{trip}
   a) check that all values are in the domain
   b) leak `input1Received` on port $leak^1_{trip}$
2. On influence `getInput` on port $infl^2_{trip}$
   a) output `getInput` on port out^2_{trip}
3. On input $(\alpha_1, a_2, b_2, \beta_{a_1}, \beta_{b_1}, \beta_{c_1}, z_2)$ on port in^2_{trip}
   a) check that all values are in the domain
   b) leak `input2Received` on port $leak^2_{trip}$
4. On influence `outputTo2` on port $infl^2_{trip}$
   a) let $a = a_1 + a_2$, $b = b_1 + b_2$, and $c = a \cdot b$
   b) let $c_1 = a_1 \cdot b_1 + a_2 \cdot b_1 + z_2 - z_1$ and $c_2 = c - c_1$
   c) let $m(a_2) = \alpha_2 \cdot a_2 + \beta_{a_2}$, $m(b_2) = \alpha_2 \cdot b_2 + \beta_{b_2}$, and $m(c_2) = \alpha_2 \cdot c_2 + \beta_{c_2}$
   d) output $\big(c_2, m(a_2), m(b_2), m(c_2)\big)$ on port out^2_{trip}
5. On input `outputTo1` on port in^2_{trip}
   a) leak `outputTo1` on port $leak^2_{trip}$
6. On influence `outputTo1` on port $infl^1_{trip}$
   a) let $m(a_1) = \alpha_1 \cdot a_1 + \beta_{a_1}$, $m(b_1) = \alpha_1 \cdot b_1 + \beta_{b_1}$, and $m(c_1) = \alpha_1 \cdot c_1 + \beta_{c_1}$
   b) output $\big(c_1, m(a_1), m(b_1), m(c_1)\big)$ on port out^1_{trip}

Figure 4.26: Triple generation functionality $\mathcal{F}^{12}_{trip}$ when both players are honest

1. On leakge `input1Received` on port $leak^1_{trip}$
   a) let $\mathcal{D}_{\alpha_2} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\alpha_2})$
   b) let $\mathcal{C}_{0_2} \leftarrow \mathsf{simencrypt}_{dom,ek_2,simtd_1}(0, r_{0_2})$
   c) send $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ to player 2
2. On receiving $(\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1})$ from player 2
   a) let $\mathcal{D}_{a_1} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{a_1})$ and $\mathcal{C}_{a_1} \leftarrow \mathsf{simeval}_{plus,ek_1,ck_1,simtd_1}(\mathcal{C}_{0_1}, 0, r_{a_1})$
   b) let $\mathcal{D}_{b_1} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{b_1})$ and $\mathcal{C}_{b_1} \leftarrow \mathsf{simeval}_{plus,ek_1,ck_1,simtd_1}(\mathcal{C}_{0_1}, 0, r_{b_1})$
   c) send $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ to player 2
3. On receiving $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ from player 2
   a) let $\mathcal{C}_{x_2} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(\mathcal{C}_{b_2}, 0, r_{a_1}, 0, r_{z_1})$
   b) send $\mathcal{C}_{x_2}$ to player 2
4. On receiving $\mathcal{C}_{x_1}$ from player 2
   a) let $\mathcal{C}_{c_1} \leftarrow \mathsf{simencrypt}_{dom,ek_1,simtd_1}(0, r_{c_1})$
   b) send $\mathcal{C}_{c_1}$ to player 2
5. On receiving $\mathcal{C}_{c_2}$ from player 2
   a) let $\mathcal{C}_{m(a_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(\mathcal{C}_{a_2}, 0, r_{\alpha_2}, 0, r_{\beta_{a_2}})$
   b) let $\mathcal{C}_{m(b_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(\mathcal{C}_{b_2}, 0, r_{\alpha_2}, 0, r_{\beta_{b_2}})$
   c) let $\mathcal{C}_{m(c_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(\mathcal{C}_{c_2}, 0, r_{\alpha_2}, 0, r_{\beta_{c_2}})$
   d) send $\left(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\right)$ to player 2
6. On receiving $\left(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\right)$ from player 2
   a) influence `outputTo1` on port $infl^1_{trip}$

Figure 4.27: Simulator $Sim^{12,1}_{trip}$ when both players are honest

1. On receiving $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ from player 1
   a) influence `getInput` on port $infl^2_{trip}$
2. On leakage `input2Received` on port $leak^2_{trip}$
   a) let $\mathcal{D}_{\alpha_1} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{\alpha_1})$
   b) let $\mathcal{C}_{0_1} \leftarrow \mathsf{simencrypt}_{dom,ek_1,simtd_2}(0, r_{0_1})$
   c) send $\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1}$ to player 1
3. On receiving $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ from player 1
   a) let $\mathcal{D}_{a_2} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{a_2})$ and $\mathcal{C}_{a_2} \leftarrow \mathsf{simeval}_{plus,ek_2,ck_2,simtd_2}(\mathcal{C}_{0_2}, 0, r_{a_2})$
   b) let $\mathcal{D}_{b_2} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{b_2})$ and $\mathcal{C}_{b_2} \leftarrow \mathsf{simeval}_{plus,ek_2,ck_2,simtd_2}(\mathcal{C}_{0_2}, 0, r_{b_2})$
   c) send $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ to player 1
4. On receiving $\mathcal{C}_{x_2}$ from player 1
   a) let $\mathcal{C}_{x_1} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(\mathcal{C}_{b_1}, 0, r_{a_2}, 0, r_{z_2})$
   b) send $\mathcal{C}_{x_1}$ to player 1
5. On receiving $\mathcal{C}_{c_1}$ from player 1
   a) let $\mathcal{C}_{c_2} \leftarrow \mathsf{simencrypt}_{dom,ek_2,simtd_2}(0)$
   b) send $\mathcal{C}_{c_2}$ to player 1
6. On receiving $\left(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\right)$ from player 1
   a) influence `outputTo2` on port $infl^2_{trip}$
7. On leakage `outputTo1` on port $leak^2_{trip}$
   a) let $\mathcal{C}_{m(a_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(\mathcal{C}_{a_1}, 0, r_{\alpha_1}, 0, r_{\beta_{a_1}})$
   b) let $\mathcal{C}_{m(b_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(\mathcal{C}_{b_1}, 0, r_{\alpha_1}, 0, r_{\beta_{b_1}})$
   c) let $\mathcal{C}_{m(c_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(\mathcal{C}_{c_1}, 0, r_{\alpha_1}, 0, r_{\beta_{c_1}})$
   d) send $\left(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\right)$ to player 1

Figure 4.28: Simulator $Sim^{12,2}_{trip}$ when both honest players are honest

1. On input $(\alpha_2, a_1, b_1, \beta_{a_2}, \beta_{b_2}, \beta_{c_2}, z_1)$ on port in^1_{trip}
   a) check that all values are in the domain
   b) leak `input1Received` on port $leak_{trip}$
2. On influence (α_1, a_2, b_2) on port $infl_{trip}$
   a) check that all values are in the domain
   b) leak $x_2 = b_2 \cdot a_1 + z_1$ on port $leak_{trip}$
3. On influence (c_2, e) on port $infl_{trip}$
   a) check that both values are in the domain
   b) let $a = a_1 + a_2$, $b = b_1 + b_2$, and $c = a \cdot b$
   c) let $c_1 = c - c_2 - e$
   d) let $m(a_2) = \alpha_2 \cdot a_2 + \beta_{a_2}$, $m(b_2) = \alpha_2 \cdot b_2 + \beta_{b_2}$, and $m(c_2) = \alpha_2 \cdot c_2 + \beta_{c_2}$
   e) leak $\big(m(a_2), m(b_2), m(c_2)\big)$ on port $leak_{trip}$
4. On influence $(\beta_{a_1}, \beta_{b_1}, \beta_{c_1})$ on port $infl_{trip}$
   a) check that all values are in the domain
   b) let $m(a_1) = \alpha_1 \cdot a_1 + \beta_{a_1}$, $m(b_1) = \alpha_1 \cdot b_1 + \beta_{b_1}$, and $m(c_1) = \alpha_1 \cdot c_1 + \beta_{c_1}$
   c) output $\big(c_1, m(a_1), m(b_1), m(c_1)\big)$ on port out^1_{trip}

Figure 4.29: Triple generation functionality $\mathcal{F}^1_{trip}$ when only player 1 is honest

1. On leakage `input1Received` on port $leak_{trip}$
   a) let $\mathcal{D}_{\alpha_2} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\alpha_2})$
   b) check $\mathcal{C}_{0_2} \leftarrow \mathsf{simencrypt}_{dom,ek_2,simtd_1}(0, r_{0_2})$
   c) send $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ to player 2
2. On receiving $(\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1})$ from player 2
   a) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{\alpha_1})$ and $\mathsf{verEncPack}_{dom,ek_1,crs_2}(\mathcal{C}_{0_1})$
   b) check $\mathsf{extractEnc}_{extd_2}(\mathcal{C}_{0_1}) = 0$
   c) let $\alpha_1 \leftarrow \mathsf{extractCom}_{extd_2}(\mathcal{D}_{\alpha_1})$
   d) let $\mathcal{D}_{a_1} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{a_1})$ and $\mathcal{C}_{a_1} \leftarrow \mathsf{simeval}_{plus,ek_1,ck_1,simtd_1}(\mathcal{C}_{0_1}, 0, r_{a_1})$
   e) let $\mathcal{D}_{b_1} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{b_1})$ and $\mathcal{C}_{b_1} \leftarrow \mathsf{simeval}_{plus,ek_1,ck_1,simtd_1}(\mathcal{C}_{0_1}, 0, r_{b_1})$
   f) send $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ to player 2
3. On receiving $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ from player 2
   a) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{a_2})$ and $\mathsf{verEvalPack}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{a_2}, \mathcal{C}_{0_2}, \mathcal{D}_{a_2})$
   b) check $\mathsf{verComPack}_{dom,ck_2,crs_2}(\mathcal{D}_{b_2})$ and $\mathsf{verEvalPack}_{plus,ek_2,ck_2,crs_2}(\mathcal{C}_{b_2}, \mathcal{C}_{0_2}, \mathcal{D}_{b_2})$
   c) let $a_2 \leftarrow \mathsf{extractCom}_{extd_2}(\mathcal{D}_{a_2})$ and $b_2 \leftarrow \mathsf{extractCom}_{extd_2}(\mathcal{D}_{b_2})$
   d) influence (α_2, a_2, b_2) on port $infl_{trip}$
4. On leakage x_2 on port $leak_{trip}$
   a) let $\mathcal{D}_{z_1} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{z_1})$
   b) let $\mathcal{C}_{x_2} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(x_2, \mathcal{C}_{b_2}, \mathcal{D}_{a_1}, \mathcal{D}_{z_1})$
   c) send $\mathcal{C}_{x_2}$ to player 2
5. On receiving $\mathcal{C}_{x_1}$ from player 2
   a) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{x_1}, \mathcal{C}_{b_1}, \mathcal{D}_{a_2}, \mathcal{D}_{z_2})$
   b) let $z_2 \leftarrow \mathsf{extractEval}_{2,extd_2}(\mathcal{C}_{x_1})$
   c) let $\mathcal{C}_{c_1} \leftarrow \mathsf{simencrypt}_{dom,ek_1,simtd_1}(0, r_{c_1})$
   d) send $\mathcal{C}_{c_1}$ to player 2
6. On receiving $\mathcal{C}_{c_2}$ from player 2
   a) check $\mathsf{verEncPack}_{dom,ek_2,crs_2}(\mathcal{C}_{c_2})$
   b) let $c_2 \leftarrow \mathsf{extractEnc}_{extd_2}(\mathcal{C}_{c_2})$
   c) let $e = a_2 \cdot b_2 + x_2 - z_2 - c_2$
   d) influence (c_2, e) on port $infl_{trip}$
7. On leakage $\big(m(a_2), m(b_2), m(c_2)\big)$ on port $leak_{trip}$
   a) let $\mathcal{D}_{\beta_{a_2}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{a_2}})$
   b) let $\mathcal{D}_{\beta_{b_2}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{b_2}})$
   c) let $\mathcal{D}_{\beta_{c_2}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{c_2}})$
   d) let $\mathcal{C}_{m(a_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(m(a_2), \mathcal{C}_{a_2}, \mathcal{D}_{\alpha_2}, \mathcal{D}_{\beta_{a_2}})$
   e) let $\mathcal{C}_{m(b_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(m(b_2), \mathcal{C}_{b_2}, \mathcal{D}_{\alpha_2}, \mathcal{D}_{\beta_{b_2}})$
   f) let $\mathcal{C}_{m(c_2)} \leftarrow \mathsf{simeval}_{multplus,ek_2,ck_1,simtd_1}(m(c_2), \mathcal{C}_{c_2}, \mathcal{D}_{\alpha_2}, \mathcal{D}_{\beta_{c_2}})$
   g) send $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ to player 2
8. On receiving $\big(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\big)$ from player 2
   a) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(a_1)}, \mathcal{C}_{a_1}, \mathcal{D}_{\alpha_1})$
   b) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(b_1)}, \mathcal{C}_{b_1}, \mathcal{D}_{\alpha_1})$
   c) check $\mathsf{verEvalPack}_{multplus,ek_1,ck_2,crs_2}(\mathcal{C}_{m(c_1)}, \mathcal{C}_{c_1}, \mathcal{D}_{\alpha_1})$
   d) let $\beta_{a_1} \leftarrow \mathsf{extractEval}_{2,extd_2}(\mathcal{C}_{m(a_1)})$
   e) let $\beta_{b_1} \leftarrow \mathsf{extractEval}_{2,extd_2}(\mathcal{C}_{m(b_1)})$
   f) let $\beta_{c_1} \leftarrow \mathsf{extractEval}_{2,extd_2}(\mathcal{C}_{m(c_1)})$
   g) influence $(\beta_{a_1}, \beta_{b_1}, \beta_{c_1})$ on port $infl_{trip}$

Figure 4.30: Simulator Sim^1_{trip} when only player 1 is honest

1. On influence α_2 on port infl_{trip}
   a) check that it is in the domain
   b) output `getInput` on port out^2_{trip}
2. On input $(\alpha_1, a_2, b_2, \beta_{a_1}, \beta_{b_1}, \beta_{c_1}, z_2)$ on port in^2_{trip}
   a) check that all values are in the domain
   b) leak `input2Received` on port leak_{trip}
3. On influence (a_1, b_1) on port infl_{trip}
   a) check that all values are in the domain
   b) leak $x_1 = b_1 \cdot a_2 + z_2$ on port leak_{trip}
4. On influence $(c_1, e, \beta_{a_2}, \beta_{b_2}, \beta_{c_2})$ on port infl_{trip}
   a) check that all values are in the domain
   b) let $a = a_1 + a_2$, $b = b_1 + b_2$, and $c = a \cdot b$
   c) let $c_2 = c - c_1 - e$
   d) let $m(a_2) = \alpha_2 \cdot a_2 + \beta_{a_2}$, $m(b_2) = \alpha_2 \cdot b_2 + \beta_{b_2}$, and $m(c_2) = \alpha_2 \cdot c_2 + \beta_{c_2}$
   e) output $\big(c_2, m(a_2), m(b_2), m(c_2)\big)$ on port out^2_{trip}
5. On input `outputTo1` on port in^2_{trip}
   a) let $m(a_1) = \alpha_1 \cdot a_1 + \beta_{a_1}$, $m(b_1) = \alpha_1 \cdot b_1 + \beta_{b_1}$, and $m(c_1) = \alpha_1 \cdot c_1 + \beta_{c_1}$
   b) leak $\big(m(a_1), m(b_1), m(c_1)\big)$ on port leak_{trip}

Figure 4.31: Triple generation functionality $\mathcal{F}^2_{trip}$ when only player 2 is honest

1. On receiving $(\mathcal{D}_{\alpha_2}, \mathcal{C}_{0_2})$ from player 1
   a) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{\alpha_2})$ and $\mathsf{verEncPack}_{dom,ek_2,crs_1}(\mathcal{C}_{0_2})$
   b) check $\mathsf{extractEnc}_{extd_1}(\mathcal{C}_{0_2}) = 0$
   c) let $\alpha_2 \leftarrow \mathsf{extractCom}_{extd_1}(\mathcal{D}_{\alpha_2})$
   d) influence α_2 on port $infl_{trip}$
2. On input `input2Received` on port $leak_{trip}$
   a) let $\mathcal{D}_{\alpha_1} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{\alpha_1})$
   b) let $\mathcal{C}_{0_1} \leftarrow \mathsf{simencrypt}_{dom,ek_1,simtd_2}(0, r_{0_1})$
   c) send $(\mathcal{D}_{\alpha_1}, \mathcal{C}_{0_1})$ to player 1
3. On receiving $(\mathcal{D}_{a_1}, \mathcal{C}_{a_1}, \mathcal{D}_{b_1}, \mathcal{C}_{b_1})$ from player 1
   a) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{a_1})$ and $\mathsf{verEvalPack}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{a_1}, \mathcal{C}_{0_1}, \mathcal{D}_{a_1})$
   b) check $\mathsf{verComPack}_{dom,ck_1,crs_1}(\mathcal{D}_{b_1})$ and $\mathsf{verEvalPack}_{plus,ek_1,ck_1,crs_1}(\mathcal{C}_{b_1}, \mathcal{C}_{0_1}, \mathcal{D}_{b_1})$
   c) let $a_1 \leftarrow \mathsf{extractCom}_{extd_1}(\mathcal{D}_{a_1})$
   d) let $b_1 \leftarrow \mathsf{extractCom}_{extd_1}(\mathcal{D}_{b_1})$
   e) let $\mathcal{D}_{a_2} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{a_2})$ and $\mathcal{C}_{a_2} \leftarrow \mathsf{simeval}_{plus,ek_2,ck_2,simtd_2}(\mathcal{C}_{0_2}, 0, r_{a_2})$
   f) let $\mathcal{D}_{b_2} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{b_2})$ and $\mathcal{C}_{b_2} \leftarrow \mathsf{simeval}_{plus,ek_2,ck_2,simtd_2}(\mathcal{C}_{0_2}, 0, r_{b_2})$
   g) send $(\mathcal{D}_{a_2}, \mathcal{C}_{a_2}, \mathcal{D}_{b_2}, \mathcal{C}_{b_2})$ to player 1
4. On receiving $\mathcal{C}_{x_2}$ from player 1
   a) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{x_2}, \mathcal{C}_{b_2}, \mathcal{D}_{a_1})$
   b) let $z_1 \leftarrow \mathsf{extractEval}_{2,extd_1}(\mathcal{C}_{x_2})$
   c) influence (a_1, b_1) on port $infl_{trip}$
5. On leakage x_1 on port $leak_{trip}$
   a) let $\mathcal{D}_{z_2} \leftarrow \mathsf{simcommit}_{dom,ck_2,simtd_2}(0, r_{z_2})$
   b) let $\mathcal{C}_{x_1} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(x_1, \mathcal{C}_{b_1}, \mathcal{D}_{a_2}, \mathcal{D}_{z_2})$
   c) send $\mathcal{C}_{x_1}$ to player 1
6. On receiving $\mathcal{C}_{c_1}$ from player 1
   a) check $\mathsf{verEncPack}_{dom,ek_1,crs_1}(\mathcal{C}_{c_1})$
   b) let $c_1 \leftarrow \mathsf{extractEnc}_{extd_1}(\mathcal{C}_{c_1})$
   c) let $\mathcal{C}_{c_2} \leftarrow \mathsf{simencrypt}_{dom,ek_2,simtd_2}(0, r_{c_2})$
   d) send $\mathcal{C}_{c_2}$ to player 1
7. On receiving $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ from player 1
   a) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(a_2)}, \mathcal{C}_{a_2}, \mathcal{D}_{\alpha_2})$
   b) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(b_2)}, \mathcal{C}_{b_2}, \mathcal{D}_{\alpha_2})$
   c) check $\mathsf{verEvalPack}_{multplus,ek_2,ck_1,crs_1}(\mathcal{C}_{m(c_2)}, \mathcal{C}_{c_2}, \mathcal{D}_{\alpha_2})$
   d) let $\beta_{a_2} \leftarrow \mathsf{extractEval}_{2,extd_1}(\mathcal{C}_{m(a_1)})$
   e) let $\beta_{b_2} \leftarrow \mathsf{extractEval}_{2,extd_1}(\mathcal{C}_{m(b_1)})$
   f) let $\beta_{c_2} \leftarrow \mathsf{extractEval}_{2,extd_1}(\mathcal{C}_{m(c_1)})$
   g) let $e = a_1 \cdot b_1 + x_1 - z_1 - c_1$
   h) influence $(c_1, e, \beta_{a_2}, \beta_{b_2}, \beta_{c_2})$ on port $infl_{trip}$
8. On leakage $\big(m(a_1), m(b_1), m(c_1)\big)$ on port $leak_{trip}$
   a) let $\mathcal{D}_{\beta_{a_1}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{a_1}})$
   b) let $\mathcal{D}_{\beta_{b_1}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{b_1}})$
   c) let $\mathcal{D}_{\beta_{c_1}} \leftarrow \mathsf{simcommit}_{dom,ck_1,simtd_1}(0, r_{\beta_{c_1}})$
   d) let $\mathcal{C}_{m(a_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(m(a_1), \mathcal{C}_{a_1}, \mathcal{D}_{\alpha_1}, \mathcal{D}_{\beta_{a_1}})$
   e) let $\mathcal{C}_{m(b_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(m(b_1), \mathcal{C}_{b_1}, \mathcal{D}_{\alpha_1}, \mathcal{D}_{\beta_{b_1}})$
   f) let $\mathcal{C}_{m(c_1)} \leftarrow \mathsf{simeval}_{multplus,ek_1,ck_2,simtd_2}(m(c_1), \mathcal{C}_{c_1}, \mathcal{D}_{\alpha_1}, \mathcal{D}_{\beta_{c_1}})$
   g) send $\big(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\big)$ to player 1

Figure 4.32: Simulator Sim^2_{trip} when only player 2 is honest

4.3 Preliminaries

Our computational model is that of the UC framework as described in [Can01]. In this model ITMs in a network communicate by writing to each others tapes, thereby passing on the right to execute. In other words, the scheduling is token-based so that any ITM may only execute when it is holding the token. Initially the special *environment* ITM $\mathcal{Z}$ holds the token. When it writes on a tape of an ITM M in the network it passes on the token and M is now allowed to execute. If the token ever gets stuck it goes back to the environment.

We say that two binary distribution ensembles X, Y are indistinguishable if for any $c, d \in \mathbb{N}$ there exists $\kappa_0 \in \mathbb{N}$ such that for all $\kappa \geq \kappa_0$ and all $z \in \cup_{k \leq \kappa^d}\{0,1\}^k$ we have $\left|\Pr[X(\kappa, z) = 1] - \Pr[Y(\kappa, z) = 1]\right| < \kappa^{-c}$. We write this as $X \approx Y$. Next, for environment $\mathcal{Z}$, adversary $\mathcal{A}$, and network N of ITMs, we write $\texttt{Exec}_{\mathcal{Z},\mathcal{A},N}(\kappa, z)$ for the random variable denoting the output bit (guess) of $\mathcal{Z}$ after interacting with $\mathcal{A}$ and N, and denote ensemble $\{\texttt{Exec}_{\mathcal{Z},\mathcal{A},N}(\kappa, z)\}_{\kappa \in \mathbb{N}, z \in \{0,1\}^\star}$ by $\texttt{Exec}_{\mathcal{Z},\mathcal{A},N}$. We may then compare networks as follows:

Definition 4.3.1 (Computational Indistinguishability)**.** *Two networks of ITMs N_1 and N_2 are* computational indistinguishability *when no probabilistic polynomial time (PPT) adversary $\mathcal{A}$ may allow a PPT environment $\mathcal{Z}$ to distinguish between them with more than negligible probability, ie. we have $\texttt{Exec}_{\mathcal{Z},\mathcal{A},N_1} \approx \texttt{Exec}_{\mathcal{Z},\mathcal{A},N_2}$. We write this as $N_1 \overset{c}{\sim} N_2$.*

By allowing different adversaries in the two networks we also obtain a notion of one network implementing another, namely network N_1 *realises* network N_2 when, for any PPT $\mathcal{A}$, there exists a PPT *simulator Sim* such that for all PPT $\mathcal{Z}$ we have $N_1 \overset{c}{\sim} N_2$.

Note that the class of environments is restricted to ensure that every execution runs in polynomial time, ie. may be simulated by a single polynomial time ITM given the security parameter κ and initial input z.

4.3.1 Commitment Scheme

A *commitment scheme* is given by PPT algorithms $\textbf{ComKeyGen}(1^\kappa) \to ck$ and $\textbf{Com}_{ck}(V, R) \to D$ for key-generation and commitment, respectively. We require that the scheme is *well-spread*, *computationally binding* and *computationally hiding*:

- **well-spread:** if no PPT adversary $\mathcal{A}$ may win the game in Fig. 4.33 with more than negligible probability (in the security parameter)
- **computationally binding:** if no PPT adversary $\mathcal{A}$ may win the game in Figure 4.34 with more than negligible probability
- **computationally hiding:** if for all PPT adversaries $\mathcal{A}$ the combination of $\mathcal{A}$ and game $\mathcal{G}_0^{com,hi}$ is indistinguishable from $\mathcal{A}$ and game $\mathcal{G}_1^{com,hi}$, as given in Figure 4.35

where, well-spread intuitively means that it is hard to predict the outcome of honestly generating a commitment.

1. Generate commitment key $ck \leftarrow \textbf{ComKeyGen}(1^\kappa)$ and send it to the adversary
2. Receive (V, D) from the adversary and check that V is a value in the domain
3. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$ and compute $D' \leftarrow \textbf{Com}_{ck}(V, R)$
4. Adversary wins if $D = D'$

Figure 4.33: Security game $\mathcal{G}^{com,ws}$ for a *well-spread* commitment scheme

1. Generate commitment key $ck \leftarrow \mathbf{ComKeyGen}(1^\kappa)$ and send it to the adversary
2. Receive (V, R) and (V', R') from the adversary and check that V and V' are values in the domain
3. Adversary wins if $\mathbf{Com}_{ck}(V, R) = \mathbf{Com}_{ck}(V', R')$ when $V \neq V'$

Figure 4.34: Security game $\mathcal{G}^{com,bin}$ for a *computationally binding* commitment scheme

1. Generate commitment key $ck \leftarrow \mathbf{ComKeyGen}(1^\kappa)$ and send it to the adversary
2. Receive (V_0, V_1) from the adversary and check that they are values in the domain
3. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$ and compute $D \leftarrow \mathbf{Com}_{ck}(V_b, R)$ for choice bit b
4. Send D to the adversary

Figure 4.35: Security game $\mathcal{G}_b^{com,hi}$ for a *computationally hiding* commitment scheme

4.3.2 Homomorphic Encryption Scheme

An *encryption scheme* is given by three PPT algorithms $\mathbf{EncKeyGen}(1^\kappa) \rightarrow (ek, dk)$, $\mathbf{Enc}_{ek}(V, R) \rightarrow C$, and $\mathbf{Dec}_{dk}(C) \rightarrow V$. An *homomorphic* encryption scheme furthermore contains a PPT algorithm $\mathbf{Eval}_{e,ek}(C_1, C_2, V_1, V_2, R) \rightarrow C$ for arithmetic expression $e(x_1, x_2, y_1, y_2)$ and randomness R for re-randomisation[22]. We require that the scheme is *well-spread*, *correct*, *history hiding* (or *formula private*), and IND-CPA secure for the entire domain:

- **well-spread:** if no PPT adversary $\mathcal{A}$ may win neither the game in Figure 4.36 nor the game in Figure 4.37 with more than negligible probability
- **correct:** if no PPT adversary $\mathcal{A}$ may win neither the game in Figure 4.38 nor the game in Figure 4.39 with more than negligible probability
- **history hiding:** if for all PPT adversaries $\mathcal{A}$ the combination of $\mathcal{A}$ and game $\mathcal{G}_0^{enc,his}$ is indistinguishable from $\mathcal{A}$ and game $\mathcal{G}_1^{enc,his}$, as given in Figure 4.40
- **IND-CPA:** if for all PPT adversaries $\mathcal{A}$ the combination of $\mathcal{A}$ and game $\mathcal{G}_0^{enc,cpa}$ is indistinguishable from $\mathcal{A}$ and game $\mathcal{G}_1^{enc,cpa}$, as given in Figure 4.41

where correct intuitively means that decryption almost always success for well-formed ciphertexts, and history hiding that a ciphertext produced using $\mathbf{Eval}_{e,ek}$ is distributed as $\mathbf{Enc}_{ek}$ on the same inputs.

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to the adversary
2. Receive (V, C) from the adversary and check that V is a value in the domain
3. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$ and compute $C' \leftarrow \mathbf{Enc}_{ek}(V, R)$
4. Adversary wins if $C = C'$

Figure 4.36: Security game $\mathcal{G}^{enc,encws}$ for a *well-spread* encryption scheme

[22] Note that the scheme needs only support the operations used by a particular protocol, ie. it is for instance not in all cases required to be fully homomorphic.

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to the adversary
2. Receive (C_1, C_2, W_1, W_2, C) from the adversary and check that W_1 and W_2 are values in the domain
3. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$ and compute $C' \leftarrow \mathbf{Eval}_{e,ek}(C_1, C_2, W_1, W_2, R)$
4. Adversary wins if $C = C'$ when $\mathbf{Dec}_{dk}(C_1) \neq \perp$ and $\mathbf{Dec}_{dk}(C_2) \neq \perp$

Figure 4.37: Security game $\mathcal{G}^{enc,evalws}$ for a *well-spread* encryption scheme

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to the adversary
2. Receive (V, R) from the adversary and check that V is a value in the domain
3. Adversary wins if $\mathbf{Dec}_{dk}(\mathbf{Enc}_{ek}(V, R)) \neq V$

Figure 4.38: Security game $\mathcal{G}^{enc,enccor}$ for a *correct* encryption scheme

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to the adversary
2. Receive (C_1, C_2, W_1, W_2, R) from the adversary and check that W_1 and W_2 are values in the domain
3. Compute $V_1 \leftarrow \mathbf{Dec}_{dk}(C_1)$ and $V_2 \leftarrow \mathbf{Dec}_{dk}(C_2)$ and check that $V_1 \neq \perp$ and $V_2 \neq \perp$
4. Adversary wins if $\mathbf{Dec}_{dk}(\mathbf{Eval}_{e,ek}(C_1, C_2, W_1, W_2, R)) \neq e(V_1, V_2, W_1, W_2)$

Figure 4.39: Security game $\mathcal{G}^{enc,evalcor}$ for a *correct* encryption scheme

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to the adversary
2. Receive (C_1, C_2, W_1, W_2) from the adversary and check that they are values in the domain
3. Compute $V_1 \leftarrow \mathbf{Dec}_{dk}(C_1)$ and $V_2 \leftarrow \mathbf{Dec}_{dk}(C_2)$ and check that $V_1 \neq \perp$ and $V_2 \neq \perp$
4. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$
5. Compute $C \leftarrow \begin{cases} \mathbf{Eval}_{e,ek}(C_1, C_2, W_1, W_2, R) & \text{if } b = 0 \\ \mathbf{Enc}_{ek}\big(e(V_1, V_2, W_1, W_2), R\big) & \text{if } b = 1 \end{cases}$ and send it to the adversary

Figure 4.40: Security game $\mathcal{G}_b^{enc,his}$ for a *history hiding* encryption scheme

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send ek to the adversary
2. Receive (V_0, V_1) from the adversary and check that they are values in the domain
3. Pick bitstring R uniformly at random from $\{0,1\}^\kappa$ and compute $C \leftarrow \mathbf{Enc}_{ek}(V_b, R)$ for choice bit b
4. Send C to the adversary

Figure 4.41: Security game $\mathcal{G}_b^{enc,cpa}$ for an IND-CPA secure encryption scheme

4.3.3 Non-Interactive Zero-Knowledge Proof-of-Knowledge Scheme

An *NIZK-PoK scheme* for binary relation $\mathcal{R}$ consists of PPT algorithms $\mathbf{CrsGen}_{\mathcal{R}}(1^\kappa) \to crs$, $\mathbf{SimCrsGen}_{\mathcal{R}}(1^\kappa) \to (crs, simtd)$, $\mathbf{ExCrsGen}_{\mathcal{R}}(1^\kappa) \to (crs, extd)$, $\mathbf{Prove}_{\mathcal{R},crs}(x, w) \to \pi$, $\mathbf{SimProve}_{\mathcal{R},simtd}(x) \to \pi$, $\mathbf{Ver}_{\mathcal{R},crs}(x, \pi) \to \{0,1\}$, and deterministic $\mathbf{Extract}_{\mathcal{R},extd}(x, \pi) \to$

w. We require that such schemes are *complete*, *computational zero-knowledge*, and *extractable*:

- **complete:** if no PPT adversary $\mathcal{A}$ may win the game in Figure 4.42 with more than negligible probability
- **computational zero-knowledge:** if for all PPT adversaries $\mathcal{A}$ the combination of $\mathcal{A}$ and game $\mathcal{G}_0^{nizk,zk}$ is indistinguishable from $\mathcal{A}$ and game $\mathcal{G}_1^{nizk,zk}$, as given in Figure 4.43
- **extractable:** if no PPT adversary $\mathcal{A}$ may win neither the game in Figure 4.44 nor the game in Figure 4.45 with more than negligible probability

and assume instantiations for:

- $\mathcal{R}_U = \{ (x,w) \,|\, D = \mathbf{Com}_{ck}(V,R) \wedge V \in U \}$ where $x = (D, ck)$ and $w = (V, R)$
- $\mathcal{R}_T = \{ (x,w) \,|\, C = \mathbf{Enc}_{ek}(V,R) \wedge V \in T \}$ where $x = (C, ek)$ and $w = (V, R)$
- $\mathcal{R}_e = \{ (x,w) \,|\, C = \mathbf{Eval}_{e,ek}(C_1, C_2, V_1, V_2, R) \wedge D_i = \mathbf{Com}_{ck}(V_i, R_i) \}$ where $x = (C, C_1, C_2, ek, D_1, D_2, ck)$ and $w = (V_1, R_1, V_2, R_2, R)$

1. Generate common reference string $crs \leftarrow \mathbf{CrsGen}(1^\kappa)$ and send it to the adversary
2. Receive (x, w) from the adversary
3. Adversary wins if $\mathbf{Ver}_{\mathcal{R},crs}(x, \mathbf{Prove}_{\mathcal{R},crs}(x,w)) = 0$ when $(x,w) \in \mathcal{R}$

Figure 4.42: Security game $\mathcal{G}^{nizk,complete}$ for a *complete* NIZK scheme

1. If $b = 0$ then generate $crs \leftarrow \mathbf{CrsGen}(1^\kappa)$; if $b = 1$ then generate $(crs, simtd) \leftarrow \mathbf{SimCrsGen}(1^\kappa)$; in both cases send crs to the adversary
2. Receive (x, w) from the adversary and check that $(x,w) \in \mathcal{R}$
3. Compute $\pi \leftarrow \begin{cases} \mathbf{Prove}_{\mathcal{R},crs}(x,w) & \text{if } b = 0 \\ \mathbf{SimProve}_{\mathcal{R},simtd}(x) & \text{if } b = 1 \end{cases}$ and send it to the adversary

Figure 4.43: Security game $\mathcal{G}_b^{nizk,zk}$ for an *computationally zero-knowledge* NIZK scheme

1. If $b = 0$ then generate $crs \leftarrow \mathbf{CrsGen}(1^\kappa)$; if $b = 1$ then generate $(crs, extd) \leftarrow \mathbf{ExCrsGen}(1^\kappa)$
2. Send crs to the adversary

Figure 4.44: Security game $\mathcal{G}_b^{nizk,exgen}$ for an *extractable* NIZK-PoK scheme

1. Generate CRS and extraction trapdoor $(crs, extd) \leftarrow \mathbf{ExCrsGen}(1^\kappa)$ and send crs to the adversary
2. Receive (x, π) from the adversary and compute $w \leftarrow \mathbf{Extract}_{\mathcal{R},extd}(x, \pi)$
3. Adversary wins if $w = \perp$ or $(x,w) \notin \mathcal{R}$

Figure 4.45: Security game $\mathcal{G}^{nizk,extract}$ for an *extractable* NIZK-PoK scheme

4.4 Real-world Interpretation

We here give a real-world computational interpretations of real and ideal protocols. First we outline the general setup of the model, followed by the description of an ITM for executing a programme P given access to an operation module implementing its available operations. Finally we give the real-world implementation of the operation modules.

4.4.1 General Structure

In the interpretation $\mathcal{RW}(Sys)$ of a system Sys each programme P is executed by ITM M_P with access to its own operation module[23] $\mathcal{O}_P$ enforcing sanity checks on received messages and implementing the operations available to P as described in Section 4.2. All messages send between these entities are annotated bitstrings BS of the following kinds: $\langle \texttt{value} : V \rangle$ and $\langle \texttt{const} : Cn \rangle$ for values and constants, $\langle \texttt{pair} : BS_1, BS_2 \rangle$ for pairings, $[\texttt{comPack} : D, ck, \pi_U, crs]$ for commitment packages, $[\texttt{encPack} : C, ek, \pi_T, crs]$ for encryption packages, and $[\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$ for evaluation packages.

The real-world model also contains a setup functionality $\mathcal{F}_{setup}$ connected to the operation modules of the cryptographic programmes. It is set to support either a real or an ideal protocol, is assumed to know the corruption scenario, and is responsible for generating and distributing the cryptographic keys and trapdoors, including leaking the public and corrupted keys to the adversary.

As an example, let Sys^{AB}_{real} be a system for some real protocol using one functionality and two authenticated channels. The real-world interpretation of it, $\mathcal{RW}(Sys^{AB}_{real})$, is then illustrated in Figure 4.46; the lines show the directional links between closed ports, and the dots represent open ports; Figure 4.47 illustrates what it had looked like had it been an ideal protocol instead. Note that we in both cases have inlined the operation modules for the plain programmes.

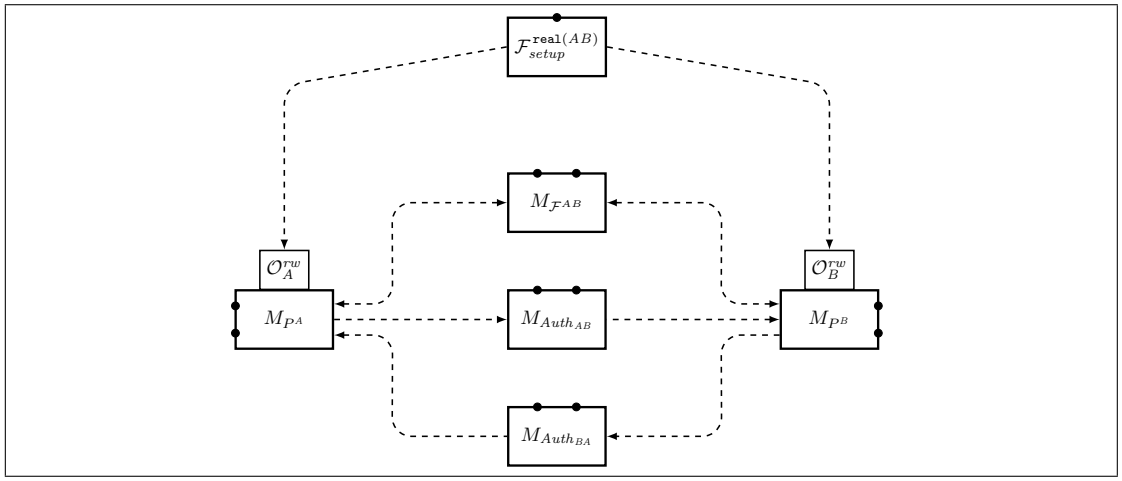

Figure 4.46: Real-world interpretation of example real protocol when both players are honest

Using ITMs defined in the remaining part of this section we may formalise the general structure as follows:

[23] We may think of an operation module simply as a functionality connected only to the owning M machine and a setup functionality. However, we use the "operation module" terminology since this makes sense in every interpretation. Furthermore, from a practical point of view it might seem arbitrary to have a separate functionality for implementing the operations as it might as well be inlined in the M machine.

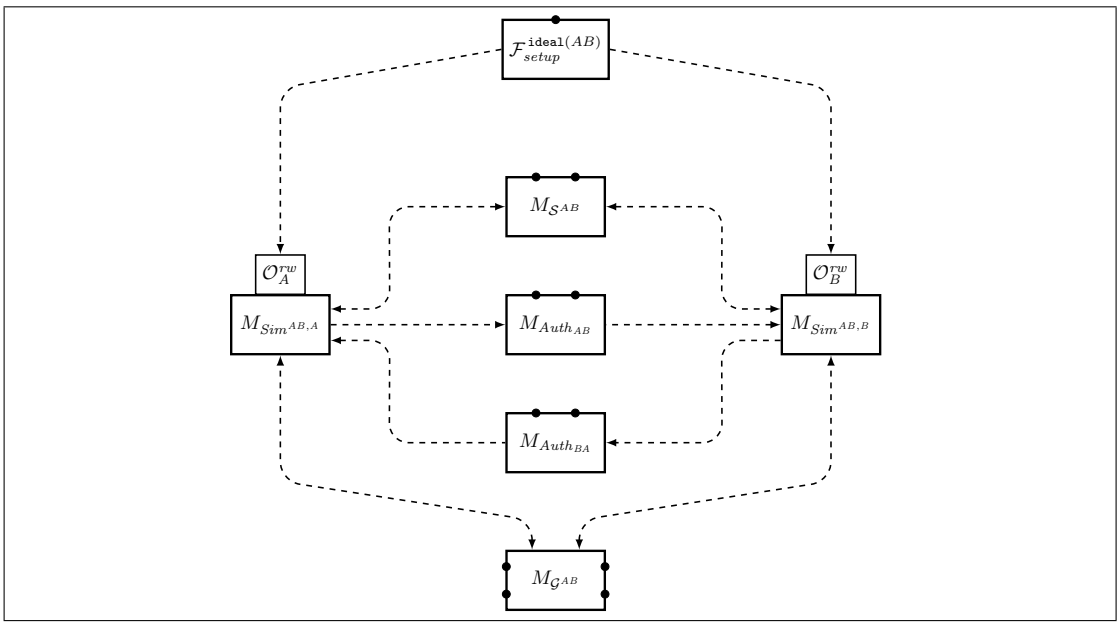

Figure 4.47: Real-world interpretation of example ideal protocol when both players are honest

Definition 4.4.1 (Real-world Interpretation)**.** *The* real-world interpretation $\mathcal{RW}(Sys)$ *of a well-formed system Sys with programmes* $P_1, \ldots, P_n$ *contains the machines* M_{P_i} *together with their operation modules* $\mathcal{O}_i$*. It also contains the setup functionality* $\mathcal{F}_{setup}$ *hardwired to match whether Sys is a real or ideal protocol.*

Besides the connections dictated by Sys, every machine M_{P_i} *is privately connected to* $\mathcal{O}_i$*. Finally,* $\mathcal{F}_{setup}$ *is privately connected with the operation modules belonging to the (at most) two cryptographic programmes.*

4.4.2 Programme Interpretation

We next describe the ITM M_P used for executing a programme P. The machine has input and output ports corresponding to the ports of P, and has access to $\mathcal{O}_P$ provinding methods

$$\texttt{storePlain}(BS) \to x, \quad \texttt{storeCrypto}_{ck_B, crs_B}(BS) \to x, \quad \texttt{retrieve}(x) \to BS$$

as well as methods corresponding to the operations available to P as outlined in Section 4.2; for instance, the methods offered to M_{P_A} for player A include

$$\texttt{commit}_{ck_A, crs_A}(v, r) \to d, \quad \texttt{encrypt}_{ek_B, crs_A}(v, r) \to c, \quad \texttt{decrypt}_{dk_A}(c) \to v$$

which all take references as input and return the same.

Informally, when a message is received by an M_P it is immediately passed to $\mathcal{O}_P$ (which, as we shall see below, among other things checks that every cryptographic package in it comes with a correct proof generated under the other player's CRS). If the message was accepted by the operation module the machine gets back a reference through which it may access the message in the future. It then executes the operations as dictated by the programme and finally either halts or sends a message to another machine.

More formally, M_P keeps in memory a position pc into programme tree P together with a set of references to randomness and messages. Initially pc points to the root of P and the set only contains randomness references (named $\mathbb{R}_P$) and message references to all values (named $\mathbb{V}$);

during execution references to received or computed messages are added and named according to the variables in the programme[24].

The execution of M_P then happens in a loop. When a message BS is received on one of its input ports p_{in} it first checks if there is an outgoing edge with port p_{in} at the node at position pc of P. If this is not the case, the message is discarded and no state is updated. Otherwise M_P asks its operation module to store it by invoking $\texttt{storePlain}(BS)$ or $\texttt{storeCrypto}_{ck,crs}(BS)$ depending on the port type. It gets back a reference if the message was accepted and `abort` if not; in the latter case the machine halts immediately.

It then names the reference x_{in} and finds the edge where ψ is satisfied. It does so by interacting with its operation module, possibly receiving temporary references in the process that are discarded when no longer needed. For instance, an edge may have condition

$$\psi = \mathsf{verEvalPack}_{sel,ek_B,ck_A,crs_A}(c, c_1, c_2, d_1, d_2)$$

checking that an incoming evaluation package BS is the result of an homomorphic evaluation on encryptions BS_1, BS_2 and values committed to by BS_1, BS_2 (the packages pointed to by the references named c, c_1, c_2, d_1 and d_2 respectively). In processing this edge M_P invokes method $\texttt{verEvalPack}_{\ldots}$ of its operation module $\mathcal{O}_P$. After finding the truth value of an edge condition, it tells $\mathcal{O}_P$ to discard all temporary references created (none in this example).

Having found the satisfied edge it continues to process the rest of the commands in the same bottom-up manner as for conditions, again discarding all temporary references and keeping only $x_1, \ldots, x_n$. For instance, an edge may have command set

$$\left\{ \frac{\mathsf{decrypt}_{dk_B}(c)}{x_v} \right\}$$

which intuitively stores the decrypted value under a reference named x_v.

Finally, M_P asks $\mathcal{O}_P$ for the message associated with the output reference named x_{out} and sends it on the output port p_{out}. The state of M_P is updated with the position of the next node along the executed edge.

4.4.3 Setup Functionality $\mathcal{F}_{setup}$

Before giving the implementation of the operation modules we describe the setup functionality that provides them with cryptographic keys through special ports $keys_A$ and $keys_B$.

The setup functionality is hardwired to operate in one of two modes, `real` or `ideal`, depending on the protocol. In both modes it generates keys for the commitment and the encryption scheme using **ComKeyGen** and **EncKeyGen**, and uses the corruption scenario to determine which decryption keys should be leaked. In mode `real` it always generates common reference strings using **CrsGen**, while in mode `ideal` it uses a mix of **SimCrsGen** and **ExCrsGen** as determined by the corruption scenario. The behaviour of $\mathcal{F}_{setup}$ is summarised in Figure 4.48.

4.4.4 Real-world Implementation of Operation Module

The final piece is describing the real-world operation modules. Each module maintains a local mapping μ between message references and bitstrings, and a local mapping ρ between randomness references and bitstrings $\{0,1\}^\kappa$ chosen uniformly at random when the module is first initialised.

[24] One implementation of this would be for the machine to also keep a mapping between variables and references. We abstract away this detail here and simply say that the variables are used to give local names to the references known by the machine.

In mode `real` on corruption scenario $\mathcal{H} \in \{AB, A, B\}$ behave as follows:

- generate ck_A and ck_B using **ComKeyGen**(1^κ)
- generate (ek_A, dk_A) and (ek_B, dk_B) using **EncKeyGen**(1^κ)
- generate crs_A and crs_B using **CrsGen**(1^κ)
- let $PK = \{ck_A, ck_B, ek_A, ek_B, crs_A, crs_B\}$ be the public keys
- send $PK \cup \{dk_{id} \mid id \in \{A, B\} \text{ is corrupt}\}$ on port $keys_{leak}$
- for honest $id \in \{A, B\}$ send $PK \cup \{dk_{id}\}$ on port $keys_{id}$

In mode `ideal` on corruption scenario $\mathcal{H} \in \{AB, A, B\}$ behave as follows:

- generate ck_A and ck_B using **ComKeyGen**(1^κ)
- generate (ek_A, dk_A) and (ek_B, dk_B) using **EncKeyGen**(1^κ)
- for honest $id \in \{A, B\}$ generate $(crs_{id}, simtd_{id})$ using **SimCrsGen**(1^κ)
- for corrupt $id \in \{A, B\}$ generate $(crs_{id}, extd_{id})$ using **ExCrsGen**(1^κ)
- let $PK = \{ck_A, ck_B, ek_A, ek_B, crs_A, crs_B\}$ be the public keys
- send $PK \cup \{dk_{id} \mid id \in \{A, B\} \text{ is corrupt}\}$ on port $keys_{leak}$
- for honest $id \in \{A, B\}$ send $PK \cup \{extd_{id'} \mid id' \text{ is corrupt}\} \cup \{simtd_{id}\}$ on port $keys_{id}$

Figure 4.48: Real-world setup functionality $\mathcal{F}_{setup}$

It also maintains a list σ of encryptions received and generated by the player associating them with their public key and their origin. This list serves the following purposes needed for the soundness result in Section 4.5.4: (i) to ensure that all encryptions in evaluation packages have the same encryption key ek since the π_e proof does not guarantee this on its own[25]; (ii) to ensure that the C_1, C_2 encryptions of evaluation packages are already known to the player[26]; and (iii) to reject certain packages that an honest player would never have produced and which cannot occur in the intermediate interpretation[27].

The methods for storing received messages are then implemented in Figure 4.49. Methods for plain operations are implemented in Figure 4.50, and for commitments, encryptions and evaluations in Figure 4.51, 4.52 and 4.53. Finally, Figure 4.54 gives the implementations for decryption and extraction operations.

[25] We need this because the intermediate implementation of eval_e fails when different encryption keys are used. This may not be the case for the **Eval**$_e$ procedure though as C_1 and C_2 are just bitstrings.

[26] This is required in order for C_1, C_2 to already have a counterpart in the intermediate model as a intermediate representation of them cannot be extracted from the evaluation package alone (unlike the D_1, D_2 commitments).

[27] One example is if it receives two evaluation packages with the same C but with, say, different D_1; an honest player would have re-randomised the result thereby with overwhelming probability not produce the same C twice. Another example is if it receives an evaluation and encryption package with the same C; again this would only happen with negligible probability if the player was honest by our well-spread assumption.

- $\texttt{storePlain}(BS) \to x$:
  1. if $\texttt{acceptPlain}(BS)$ returns $\texttt{false}$ then return $\texttt{abort}$
  2. otherwise pick a fresh reference x; store $\mu(x) \mapsto BS$; and return x
- $\texttt{storeCrypto}_{ck,crs}(BS) \to x$:
  1. if $\texttt{acceptCrypto}_{ck,crs}(BS)$ returns $\texttt{false}$ then return $\texttt{abort}$
  2. otherwise pick a fresh reference x; store $\mu(x) \mapsto BS$; and return x
- $\texttt{acceptPlain}(BS) \to \{\texttt{true}, \texttt{false}\}$:
  - BS match $\langle \texttt{value} : V \rangle$: verify that V may be parsed as a value; return $\texttt{true}$
  - BS match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return $\texttt{true}$
  - BS match $\langle \texttt{pair} : BS_1, BS_2 \rangle$: verify $\texttt{acceptPlain}(BS_1)$ and $\texttt{acceptPlain}(BS_2)$; return $\texttt{true}$
  - return $\texttt{false}$ if none of the above apply or if any verification fails
- $\texttt{acceptCrypto}_{ck,crs}(BS) \to \{\texttt{true}, \texttt{false}\}$:
  - BS match $\langle \texttt{value} : V \rangle$: verify that V may be parsed as a value; return $\texttt{true}$
  - BS match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return $\texttt{true}$
  - BS match $\langle \texttt{pair} : BS_1, BS_2 \rangle$: verify $\texttt{acceptCrypto}_{ck,crs}(BS_1)$ and $\texttt{acceptCrypto}_{ck,crs}(BS_2)$; return $\texttt{true}$
  - BS match $[\texttt{comPack} : D, ck, \pi_U, crs]$: verify that π_U is a valid proof under crs of type U for D, ck by running $\mathbf{Ver}_{U,crs}(D, ck, \pi_U)$; return $\texttt{true}$
  - BS match $[\texttt{encPack} : C, ek, \pi_T, crs]$ with $ek \in \{ek_A, ek_B\}$: verify that π_T is a valid proof under crs of type T for C, ek by running $\mathbf{Ver}_{T,crs}(C, ek, \pi_T)$; check that $\sigma(C, ek) \in \{\bot, \texttt{encother}\}$; update $\sigma(C, ek) \mapsto \texttt{encother}$; return $\texttt{true}$
  - BS match $[\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$ with $ek \in \{ek_A, ek_B\}$: verify that π_e is a valid proof under crs of expression e for $C_1, \ldots, ck$ by running $\mathbf{Ver}_{e,crs}(C, C_1, C_2, ek, D_1, D_2, ck, \pi_e)$; verify that C_1 and C_2 are already known by the programme and have the proper key by checking $\sigma(C_1, ek) \neq \bot$ and $\sigma(C_2, ek) \neq \bot$; verify that C has not been defined before by checking $\sigma(C, ek) \neq \bot \implies \sigma(C, ek) = \texttt{evalother}(BS)$; update $\sigma(C, ek) \mapsto \texttt{evalother}(BS)$; return $\texttt{true}$
  - return $\texttt{false}$ if none of the above apply or if any verification fails
- $\texttt{retrieve}(x) \to BS$: return $\mu(x)$

Figure 4.49: Real-world implementation of well-formed checks for programmes

- $\texttt{isConst}(x) \to B$: if $\mu(x)$ matches with $\langle \texttt{const} : \dots \rangle$ then return `true` else `false`
- $\texttt{eqConst}_{Cn}(x) \to B$: if $\mu(x)$ matches with $\langle \texttt{const} : Cn \rangle$ then return `true` else `false`
- $\texttt{isValue}(v) \to B$: if $\mu(v)$ matches with $\langle \texttt{value} : \dots \rangle$ then return `true` else `false`
- $\texttt{eqValue}(v_1, v_2) \to B$: for $i \in [2]$ match $\mu(v_i)$ with $\langle \texttt{value} : V_i \rangle$; return `true` if $V_1 = V_2$ else `false`
- $\texttt{inType}_U(v) \to B$: match $\mu(v)$ with $\langle \texttt{value} : V \rangle$; return `true` if V is in type U else `false`
- $\texttt{inType}_T(v) \to B$: match $\mu(v)$ with $\langle \texttt{value} : V \rangle$; return `true` if V is in type T else `false`
- $\texttt{peval}_f(v_1, v_2, w_1, w_2) \to v$: for $i \in [2]$ match $\mu(v_i)$ with $\langle \texttt{value} : V_i \rangle$ and $\mu(w_i)$ with $\langle \texttt{value} : W_i \rangle$; evaluate expression f on these values, ie. let $V = f(V_1, V_2, W_1, W_2)$; pick fresh reference v, update $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{isPair}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{pair} : \dots \rangle$ then return `true` else `false`
- $\texttt{pair}(x_1, x_2) \to x$: pick fresh reference x, update $\mu(v) \mapsto \langle \texttt{pair} : \mu(x_1), \mu(x_2) \rangle$, and return x
- $\texttt{first}(x) \to x_1$: if $\mu(x)$ match with $\langle \texttt{pair} : BS_1, BS_2 \rangle$ then pick fresh reference x_1, update $\mu(x_1) \mapsto BS_1$, and return x_1; else return `abort`
- $\texttt{second}(x) \to x_2$: if $\mu(x)$ match with $\langle \texttt{pair} : BS_1, BS_2 \rangle$ then pick fresh reference x_2, update $\mu(x_2) \mapsto BS_2$, and return x_2; else return `abort`

Figure 4.50: Real-world implementation of plain operations for programmes

- $\texttt{isComPack}(x) \to B$: if $\mu(x)$ match with $[\texttt{comPack} : \dots]$ then return `true` else `false`
- $\texttt{verComPack}_{U,ck,crs}(d) \to B$: if $\mu(d)$ match with $[\texttt{comPack} : -, ck, \pi_U, crs]$ return `true` else `false`
- $\texttt{commit}_{U,ck,crs}(v, r) \to d$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$ and check that V is in type U
  2. let $R = \rho(r)$ be the randomness associated with r
  3. compute $D \leftarrow \mathbf{Com}_{ck}(V, R)$ and $\pi_U \leftarrow \mathbf{Prove}_{U,crs}(D, ck, V, R)$
  4. pick fresh reference d, update $\mu(d) \mapsto [\texttt{comPack} : D, ck, \pi_U, crs]$, and return d
- $\texttt{simcommit}_{U,ck,simtd}(v, r) \to d$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  2. let $R = \rho(r)$ be the randomness associated with r
  3. let crs be the CRS corresponding to $simtd$
  4. compute $D \leftarrow \mathbf{Com}_{ck}(V, R)$ and $\pi_U \leftarrow \mathbf{SimProve}_{U,simtd}(D, ck)$
  5. pick fresh reference d, update $\mu(d) \mapsto [\texttt{comPack} : D, ck, \pi_U, crs]$, and return d

Figure 4.51: Real-world implementation of commitment operations for programmes

- $\texttt{isEncPack}(x) \to B$: if $\mu(x)$ match with $[\texttt{encPack} : \ldots]$ then return `true` else `false`
- $\texttt{verEncPack}_{T,ek,crs}(x) \to B$: if $\mu(c)$ match with $[\texttt{encPack} : -, ek, \pi_T, crs]$ then return `true` else `false`
- $\texttt{encrypt}_{T,ek,crs}(v, r) \to c$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$ and check that V is in type T
  2. let $R = \rho(r)$ be the randomness associated with r
  3. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$ and $\pi_T \leftarrow \mathbf{Prove}_{T,crs}(C, ek, V, R)$
  4. update $\sigma(C, ek) \mapsto \texttt{encme}$
  5. pick fresh reference c, update $\mu(c) \mapsto [\texttt{encPack} : C, ek, \pi_T, crs]$, and return c
- $\texttt{simencrypt}_{T,ek,simtd}(v, r) \to c$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  2. let $R = \rho(r)$ be the randomness associated with r
  3. let crs be the CRS corresponding to $simtd$
  4. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$ and $\pi_T \leftarrow \mathbf{SimProve}_{T,simtd}(C, ek)$
  5. update $\sigma(C, ek) \mapsto \texttt{encme}$
  6. pick fresh reference c, update $\mu(c) \mapsto [\texttt{encPack} : C, ek, \pi_T, crs]$, and return c

Figure 4.52: Real-world implementation of encryption operations for programmes

- $\texttt{isEvalPack}(x) \to B$: if $\mu(x)$ match with $[\texttt{evalPack} : \dots]$ then return `true` else `false`
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2) \to B$:
  1. for $i \in [2]$ match $\mu(c_i)$ with $[\texttt{encPack} : C_i, ek, \dots]$ or $[\texttt{evalPack} : C_i, ek, \dots]$
  2. match $\mu(c)$ with $[\texttt{evalPack} : -, C_1, C_2, ek, -, -, ck, \pi_e, crs]$
  3. return `true` if successful else `false`
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2, d_1, d_2) \to B$:
  1. for $i \in [2]$ match $\mu(c_i)$ with $[\texttt{encPack} : C_i, ek, \dots]$ or $[\texttt{evalPack} : C_i, ek, \dots]$
  2. for $i \in [2]$ match $\mu(d_i)$ with $[\texttt{comPack} : D_i, ck, \dots]$
  3. match $\mu(c)$ with $[\texttt{evalPack} : -, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$
  4. return `true` if successful else `false`
- $\texttt{eval}_{e,ek,ck,crs}(c_1, c_2, v_1, r_1, v_2, r_2) \to c$:
  1. for $i \in [2]$ match $\mu(c_i)$ with $[\texttt{encPack} : C_i, ek, \dots]$ or $[\texttt{evalPack} : C_i, ek, \dots]$
  2. for $i \in [2]$ match $\mu(v_i)$ with $\langle \texttt{value} : V_i \rangle$
  3. for $i \in [2]$ compute $D_i \leftarrow \mathbf{Com}_{ck}(V_i, R_i)$
  4. pick fresh randomness $R \in \{0,1\}^\kappa$
  5. compute $C \leftarrow \mathbf{Eval}_{e,ek}(C_1, C_2, V_1, V_2, R)$
  6. compute $\pi_e \leftarrow \mathbf{Prove}_{e,crs}(C, C_1, C_2, ek, D_1, D_2, ck, V_1, R_1, V_2, R_2)$
  7. update $\sigma(C, ek) \mapsto \texttt{evalme}$
  8. pick fresh reference c, update $\mu(c) \mapsto [\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, and return c
- $\texttt{simeval}_{e,ek,ck,simtd}(c_1, c_2, v_1, r_1, v_2, r_2) \to c$:
  - (as $\texttt{eval}_{e,ek,ck,crs}$ but using $\mathbf{SimProve}_{e,simtd}$ instead of $\mathbf{Prove}_{e,crs}$)
- $\texttt{simeval}_{e,ek,ck,simtd}(v, c_1, c_2, d_1, d_2) \to c$:
  1. for $i \in [2]$ match $\mu(c_i)$ with $[\texttt{encPack} : C_i, ek, \dots]$ or $[\texttt{evalPack} : C_i, ek, \dots]$
  2. for $i \in [2]$ match $\mu(d_i)$ with $[\texttt{comPack} : D_i, ck, \dots]$
  3. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  4. pick fresh randomness $R \in \{0,1\}^\kappa$
  5. let crs be the CRS corresponding to $simtd$
  6. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$
  7. compute $\pi_e \leftarrow \mathbf{SimProve}_{e,simtd}(C, C_1, C_2, ek, D_1, D_2, ck)$
  8. update $\sigma(C, ek) \mapsto \texttt{evalme}$
  9. pick fresh reference c, update $\mu(c) \mapsto [\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, and return c

Figure 4.53: Real-world implementation of evaluation operations for programmes

- $\texttt{decrypt}_{dk}(c) \to v$:
  1. match $\mu(c)$ with $[\texttt{encPack} : C, ek, \dots]$ or $[\texttt{evalPack} : C, ek, \dots]$
  2. check that dk is the decryption key for ek
  3. compute $V \leftarrow \mathbf{Dec}_{dk}(C)$
  4. pick fresh reference v, update $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractCom}_{extd}(d) \to v$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\mu(d)$ with $[\texttt{comPack} : D, ck, \pi_U, crs]$
  3. compute $(V, R) \leftarrow \mathbf{Extr}_{U,extd}(D, ck, \pi_U)$
  4. pick fresh reference v, update $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractEnc}_{extd}(c) \to v$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\mu(c)$ with $[\texttt{encPack} : C, ek, \pi_T, crs]$
  3. compute $(V, R) = \mathbf{Extr}_{T,extd}(C, ek, \pi_T)$
  4. pick fresh reference v, update $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractEval}_{1,extd}(c) \to v_1$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\mu(c)$ with $[\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$
  3. compute $(V_1, R_1, V_2, R_2, R) \leftarrow \mathbf{Extr}_{e,extd}(C, C_1, C_1, ek, D_1, D_2, ck, \pi_e)$
  4. pick fresh reference v_1, update $\mu(v_1) \mapsto \langle \texttt{value} : V_1 \rangle$, and return v_1
- $\texttt{extractEval}_{2,extd}(c) \to v_2$:
  - (as $\texttt{extractEval}_{1,extd}$ but returning V_2 instead of V_1)

Figure 4.54: Real-world implementation of decryption and extraction operations for programmes

4.5 Intermediate Interpretation

This section gives the computational interpretation used as an intermediate step in linking the real-world interpretation with the symbolic interpretation given in Section 4.6. The main difference in this model is that the cryptographic primitives are replaced with a global memory to which the adversary only has restricted access in the form of a fixed set of methods.

4.5.1 General Structure

The intermediate interpretation has a number of similarities with the real-world interpretation: besides having the same underlying computation model, the M_P machines for executing programmes are also identical. However, the operation modules $\mathcal{O}_P$ are different and the setup functionality is replaced with a global memory accessible only to the operation modules and $\mathcal{O}_{adv}$ offering access to the adversary through a fixed set of methods[28]. If we put all the modules together with the global memory we may think of them as simply a functionality $\mathcal{F}_{aux}$, meaning that protocols in the intermediate interpretation are running in a $\mathcal{F}_{aux}$-hybrid model.

The basic principle is that all cryptographic messages passed around among the entities are bitstrings drawn uniformly at random from $\{0,1\}^\kappa$, dubbed *handles*, and ranged over by H. They are associated to *data objects* in the global memory that hold the plaintext values: commitment objects take form $(\texttt{com} : V, R, ck)$; encryption objects $(\texttt{enc} : V, R, ek)$; proof objects[29] $(\texttt{proof}_U : H_D, ck, crs)$, $(\texttt{proof}_T : H_C, ek, crs)$, and $(\texttt{proof}_e : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, crs)$; and finally packages by objects $(\texttt{comPack} : H_D, ck, H_\pi, crs)$, $(\texttt{encPack} : H_C, ek, H_\pi, crs)$, and $(\texttt{evalPack} : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, H_\pi, crs)$. The ck,ek,crs here are simply fixed constants used to indicate the creator and owner of the objects. Values, constants, and pairing are not stored in the global memory but instead encoded as in the real-world interpretation, yet we shall also use H to range over these.

Definition 4.5.1 (Intermediate Interpretation)**.** *The* intermediate interpretation $\mathcal{I}(Sys)$ *of a well-formed system Sys with programmes* $P_1, \ldots, P_n$ *contains the machines* M_{P_i} *together with their operation modules* $\mathcal{O}_i$*. It also contains the operation module* $\mathcal{O}_{adv}$ *given to the adversary and the global memory functionality* $\mathcal{F}_{mem}$*.*

Besides the connections dictated by Sys, every machine M_{P_i} *is privately connected to* $\mathcal{O}_i$*, and every* $\mathcal{O}_i$*, including* $\mathcal{O}_{adv}$*, is in turn privately connected to* $\mathcal{F}_{mem}$*.*

4.5.2 Intermediate Interpretation of Operation Modules

The operation modules follow their real-world counterpart somewhat straight-forwardly, yet operates on the data object in the global memory instead of using the procedures of the primitives. They still keep a local mapping ρ from randomness references r to random bitstrings R drawn uniformly at random from $\{0,1\}^\kappa$ at initialisation, and they still have a local memory μ between references and messages.

The various implementations are given in Figure 4.55, 4.56, 4.57, 4.58 and 4.59 where γ denotes the global memory (recall that H is used to range over both random handles, and pairings of these with values and constants). Note that some guarantees are now provided by the model itself as a consequence of the adversary being limited in what he may do; for

[28] An implication of this model is that every message is already somewhat well-formed in the sense that it is either garbage or correctly generated through a method invocation.

[29] Note that proof objects do not have a randomness (or counter) component, meaning that there cannot be several different proof objects for the same public parameters; intuitively it makes no difference as there is no operation for programmes to check equality of proofs and packages, and since we do not allow programmes to send back received packages. We have gone with this option to simplify the symbolic model but may easily remove it and obtain the same results.

- $\texttt{storePlain}(H) \to x$:
  1. if $\texttt{acceptPlain}(H)$ returns `false` then return `abort`
  2. otherwise pick a fresh reference x, store $\mu(x) \mapsto H$, and return x
- $\texttt{storeCrypto}_{ck,crs}(H) \to x$:
  1. if $\texttt{acceptCrypto}_{ck,crs}(H)$ returns `false` then return `abort`
  2. otherwise pick a fresh reference x, store $\mu(x) \mapsto H$, and return x
- $\texttt{acceptPlain}(H) \to B$:
  - H match $\langle \texttt{value} : V \rangle$: verify that V may be parsed as a value; return `true`
  - H match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return `true`
  - H match $\langle \texttt{pair} : H_1, H_2 \rangle$: verify $\texttt{acceptPlain}(H_1)$ and $\texttt{acceptPlain}(H_2)$; return `true`
  - return `false` if none of the above apply or if any verification fails
- $\texttt{acceptCrypto}_{ck,crs}(H) \to B$:
  - H match $\langle \texttt{value} : V \rangle$: verify that V may be parsed as a value; return `true`
  - H match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return `true`
  - H match $\langle \texttt{pair} : H_1, H_2 \rangle$: verify $\texttt{acceptCrypto}_{ck,crs}(H_1)$ and $\texttt{acceptCrypto}_{ck,crs}(H_2)$; return `true`
  - $\gamma(H)$ match $(\texttt{comPack} : H_D, ck, H_\pi, crs)$: verify that $\gamma(H_\pi)$ match $(\texttt{proof}_U : H_D, ck, crs)$; return `true`
  - $\gamma(H)$ match $(\texttt{encPack} : H_C, ek, H_\pi, crs)$ with $ek \in \{ek_A, ek_B\}$: verify that $\gamma(H_\pi)$ match $(\texttt{proof}_T : H_C, ek, crs)$; update $\sigma(H_C) \mapsto \texttt{ok}$; return `true`
  - $\gamma(H)$ match $(\texttt{evalPack} : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, H_\pi, crs)$ with $ek \in \{ek_A, ek_B\}$: verify that $\gamma(H_\pi)$ match $(\texttt{proof}_e : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, crs)$; verify that H_{C_1} and H_{C_2} are already known by the party by checking $\sigma(H_{C_1}) = \texttt{ok}$ and $\sigma(H_{C_2}) = \texttt{ok}$; update $\sigma(H_C) \mapsto \texttt{ok}$; return `true`
  - return `false` if none of the above apply or if any verification fails
- $\texttt{retrieve}(x) \to H$: refresh all object handles in $\mu(x)$ and return the result

Figure 4.55: Intermediate implementation of storing etc. for programmes

instance, it is not possible for the adversary to construct packages with an invalid proof, and even adversarial evaluated ciphertexts are correctly re-randomised. This justifies the fact that less conditions are enforced using the σ list, and that in this model it is only needed to ensure that an evaluation package received by a player is rejected if its sub-encryptions C_1, C_2 have not already been received.

- $\texttt{isConst}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{const} : \dots \rangle$ then return `true` else `false`
- $\texttt{eqConst}_{Cn}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{const} : Cn \rangle$ return `true` else `false`
- $\texttt{isValue}(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : \dots \rangle$ return `true` else `false`
- $\texttt{eqValue}(v_1, v_2) \to B$: if $\mu(v_i)$ match $\langle \texttt{value} : V_i \rangle$ for $i \in [2]$, and $V_1 = V_2$, return `true` else `false`
- $\texttt{inType}_U(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : V \rangle$, and V is in type U, return `true` else `false`
- $\texttt{inType}_T(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : V \rangle$, and V is in type T, return `true` else `false`
- $\texttt{peval}_f(v_1, v_2, w_1, w_2) \to v$: match $\mu(v_i)$ with $\langle \texttt{value} : V_i \rangle$ and $\mu(w_i)$ with $\langle \texttt{value} : W_i \rangle$; evaluate f on these values, ie. let $V = f(V_1, V_2, W_1, W_2)$; pick a fresh reference v, store $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{isPair}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{pair} : \dots \rangle$ then return `true` else `false`
- $\texttt{pair}(x_1, x_2) \to x$: pick a fresh reference x, store $\mu(x) \mapsto \langle \texttt{pair} : \mu(x_1), \mu(x_2) \rangle$, and return x
- $\texttt{first}(x) \to x_1$: match $\mu(x)$ with $\langle \texttt{pair} : H_1, H_2 \rangle$; pick fresh reference x_1, store $\mu(x_1) \mapsto H_1$, and return x_1
- $\texttt{second}(x) \to x_2$: match $\mu(x)$ with $\langle \texttt{pair} : H_1, H_2 \rangle$; pick fresh reference x_2, store $\mu(x_2) \mapsto H_2$, and return x_2

Figure 4.56: Intermediate implementation of plain operations

- $\texttt{isComPack}(x) \to B$: if $\gamma(\mu(x))$ match with $(\texttt{comPack} : \dots)$ then return `true` else `false`
- $\texttt{verComPack}_{U,ck,crs}(d) \to B$: if $\gamma(\mu(d))$ match with $(\texttt{comPack} : -, ck, H_\pi, crs)$ and $\gamma(H_\pi)$ match with $(\texttt{proof}_U : \dots)$ then return `true` else `false`
- $\texttt{commit}_{U,ck,crs}(v, r) \to d$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$ and check that V is in type U
  2. let $R = \rho(r)$ be the randomness associated with r
  3. pick handle H_D uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_D) \mapsto (\texttt{com} : V, R, ck)$
  4. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_U : H_D, ck, crs)$
  5. pick handle H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{comPack} : H_D, ck, H_\pi, crs)$
  6. pick fresh reference d, store $\mu(d) \mapsto H$, and return d
- $\texttt{simcommit}_{U,ck,simtd}(v, r) \to d$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  2. let $R = \rho(r)$ be the randomness associated with r
  3. let crs be the CRS corresponding to $simtd$
  4. pick handle H_D uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_D) \mapsto (\texttt{com} : V, R, ck)$
  5. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_U : H_D, ck, crs)$
  6. pick handle H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{comPack} : H_D, ck, H_\pi, crs)$
  7. pick fresh reference d, store $\mu(d) \mapsto H$, and return d

Figure 4.57: Intermediate implementation of commitment packages

- $\texttt{isEncPack}(x) \to B$: if $\gamma(\mu(x))$ match with $(\texttt{encPack} : \dots)$ return `true` else `false`
- $\texttt{verEncPack}_{T,ek,crs}(c) \to B$: if $\gamma(\mu(c))$ match with $(\texttt{encPack} : -, ek, H_\pi, crs)$ and $\gamma(H_\pi)$ with $(\texttt{proof}_T : \dots)$ return `true` else `false`
- $\texttt{encrypt}_{T,ek,crs}(v, r) \to c$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$ and check that V is in type T
  2. let $R = \rho(r)$ be the randomness associated with r
  3. pick handle H_C uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_C) \mapsto (\texttt{enc} : V, R, ek)$
  4. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_T : H_C, ek, crs)$
  5. pick handle H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{encPack} : H_C, ek, H_\pi, crs)$
  6. update $\sigma(H_C) \mapsto \texttt{ok}$
  7. pick fresh reference c, store $\mu(c) \mapsto H$, and return c
- $\texttt{simencrypt}_{T,ek,simtd}(v, r) \to c$:
  1. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  2. let crs be the CRS corresponding to $simtd$
  3. let $R = \rho(r)$ be the randomness associated with r
  4. pick handle H_C uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_C) \mapsto (\texttt{enc} : V, R, ek)$
  5. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_T : H_C, ek, crs)$
  6. pick handle H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{encPack} : H_C, ek, H_\pi, crs)$
  7. update $\sigma(H_C) \mapsto \texttt{ok}$
  8. pick fresh reference c, store $\mu(c) \mapsto H$, and return c

Figure 4.58: Intermediate implementation of encryption packages

- $\texttt{isEvalPack}(x) \to B$: if $\gamma(\mu(x))$ match with $(\texttt{evalPack} : \dots)$ return `true` else `false`
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2) \to B$:
  1. for $i \in [2]$ match $\gamma(\mu(c_i))$ with $(\texttt{encPack} : H_{C_i} \dots)$ or $(\texttt{evalPack} : H_{C_i} \dots)$
  2. match $\gamma(\mu(c))$ with $(\texttt{evalPack} : -, H_{C_1}, H_{C_2}, ek, -, -, ck, H_\pi, crs)$ and $\gamma(H_\pi)$ with $(\texttt{proof}_e : \dots)$
  3. if all successful return `true` else `false`
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2, d_1, d_2) \to B$:
  1. for $i \in [2]$ match $\gamma(\mu(c_i))$ with $(\texttt{encPack} : H_{C_i} \dots)$ or $(\texttt{evalPack} : H_{C_i} \dots)$
  2. for $i \in [2]$ match $\gamma(\mu(d_i))$ with $(\texttt{comPack} : H_{D_i} \dots)$
  3. match $\gamma(\mu(c))$ with $(\texttt{evalPack} : -, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, H_\pi, crs)$ and $\gamma(H_\pi)$ with $(\texttt{proof}_e : \dots)$
  4. if all successful return `true` else `false`
- $\texttt{eval}_{e,ek,ck,simtd}(c_1, c_2, w_1, r_1, w_2, r_2) \to c$:
  1. for $i \in [2]$ match $\gamma(\mu(c_i))$ with $(\texttt{encPack} : H_{C_i}, ek \dots)$ or $(\texttt{evalPack} : H_{C_i}, -, ek \dots)$
  2. for $i \in [2]$ match $\gamma(H_{C_i})$ with $(\texttt{enc} : V_i \dots)$ and $\mu(w_i)$ with $\langle \texttt{value} : W_i \rangle$
  3. for $i \in [2]$ let $R_i = \rho(r_i)$ be the randomness associated with r_i
  4. for $i \in [2]$ pick handle H_{D_i} uniformly at random from $\{0,1\}^\kappa$, store $\gamma(H_{D_i}) \mapsto (\texttt{com} : W_i, R_i, ck)$
  5. pick randomness R uniformly at random from $\{0,1\}^\kappa$
  6. let $V = e(V_1, V_2, W_1, W_2)$ be the evaluation of e
  7. pick handle H_C uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_C) \mapsto (\texttt{enc} : V, R, ek)$
  8. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_e : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ek, crs)$
  9. pick handles H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{evalPack} : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, H_\pi, crs)$
  10. update $\sigma(H_C) \mapsto \texttt{ok}$
  11. pick fresh reference c, store $\mu(c) \mapsto H$, and return c
- $\texttt{simeval}_{e,ek,ck,simtd}(c_1, c_2, w_1, r_1, w_2, r_2) \to c$:
  - (as $\texttt{eval}_{e,ek,ck,crs}$)
- $\texttt{simeval}_{e,ek,ck,simtd}(v, c_1, c_2, d_1, d_2) \to c$:
  1. for $i \in [2]$ match $\gamma(\mu(c_i))$ with $(\texttt{encPack} : H_{C_i}, ek \dots)$ or $(\texttt{evalPack} : H_{C_i}, -, ek \dots)$
  2. for $i \in [2]$ match $\gamma(\mu(d_i))$ with $(\texttt{comPack} : H_{D_i}, ck \dots)$
  3. match $\mu(v)$ with $\langle \texttt{value} : V \rangle$
  4. let crs be the CRS corresponding to $simtd$
  5. pick fresh randomness R uniformly at random from $\{0,1\}^\kappa$
  6. pick handle H_C uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_C) \mapsto (\texttt{enc} : V, R, ek)$
  7. pick handle H_π uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H_\pi) \mapsto (\texttt{proof}_e : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, crs)$
  8. pick handles H uniformly at random from $\{0,1\}^\kappa$ and store $\gamma(H) \mapsto (\texttt{evalPack} : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, H_\pi, crs)$
  9. update $\sigma(H_C) \mapsto \texttt{ok}$
  10. pick fresh reference c, store $\mu(c) \mapsto H$, and return c

Figure 4.59: Intermediate implementation of evaluation packages

- $\texttt{decrypt}_{dk}(c) \to v$:
  1. let ek be the encryption key corresponding to dk
  2. match $\gamma(\mu(c))$ with $(\texttt{encPack} : H_C, ek, \dots)$ or $(\texttt{evalPack} : H_C, ek, \dots)$
  3. match H_C with $(\texttt{enc} : V, \dots)$
  4. pick fresh reference v, store $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractCom}_{extd}(d) \to v$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\gamma(\mu(d))$ with $(\texttt{comPack} : \dots, H_\pi, crs)$
  3. match $\gamma(H_\pi)$ with $(\texttt{proof}_U : (\texttt{com} : V, \dots), \dots)$
  4. pick fresh reference v, store $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractEnc}_{extd}(c) \to v$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\gamma(\mu(c))$ with $(\texttt{encPack} : \dots, H_\pi, crs)$
  3. match $\gamma(H_\pi)$ with $(\texttt{proof}_T : (\texttt{enc} : V, \dots), \dots)$
  4. pick fresh reference v, store $\mu(v) \mapsto \langle \texttt{value} : V \rangle$, and return v
- $\texttt{extractEval}_{1,extd}(c) \to v_1$:
  1. let crs be the CRS corresponding to $extd$
  2. match $\gamma(\mu(c))$ with $(\texttt{evalPack} : \dots, H_\pi, crs)$
  3. match $\gamma(H_\pi)$ with $(\texttt{proof}_e : \dots, (\texttt{com} : V_1, \dots), (\texttt{com} : V_2, \dots), \dots)$
  4. pick fresh reference v_1, store $\mu(v_1) \mapsto \langle \texttt{value} : V_1 \rangle$, and return v_1
- $\texttt{extractEval}_{2,extd}(c) \to v_2$:
  - (as $\texttt{extractEval}_{1,extd}$ but returning V_2 instead of V_1)

Figure 4.60: Intermediate implementation of decryption and extraction operations

$\texttt{isConst}(H) \to B$ $\quad$ $\texttt{pair}(H_1, H_2) \to H$

$\texttt{isValue}(H) \to B$ $\quad$ $\texttt{first}(H) \to H_1$

$\texttt{isPair}(H) \to B$ $\quad$ $\texttt{second}(H) \to H_2$

$\texttt{isComPack}(H) \to B$ $\quad$ $\texttt{verComPack}_U(H) \to B$

$\texttt{isEncPack}(H) \to B$ $\quad$ $\texttt{verEncPack}_T(H) \to B$

$\texttt{isEvalPack}(H) \to B$ $\quad$ $\texttt{verEvalPack}_e(H) \to B$

$\texttt{commit}_{U,ck,crs}(V, R) \to H$ $\quad$ $\texttt{eval}_{e,ck,ek,crs}(H_{C_1}, H_{C_2}, V_1, R_1, V_2, R_2) \to H$

$\texttt{encrypt}_{T,ek,crs}(V, R) \to H$ $\quad$ $\texttt{decrypt}_{dk}(H_C) \to V$

$\texttt{comOf}(H) \to H_D$ gives handle to commitment object H_D of commitment package H

$\texttt{encOf}(H) \to H_C$ gives handle to encryption object H_C of enc. or eval. package H

$\texttt{encOf}_i(H) \to H_{C_i}$ gives handle to ith encryption object H_{C_i} of evaluation package H

$\texttt{comOf}_i(H) \to H_{D_i}$ gives handle to ith commitment object H_{D_i} of evaluation package H

$\texttt{isCkOf}_{ck}(H) \to B$ indicates if ck is the commitment key used by package H

$\texttt{isEkOf}_{ek}(H) \to B$ indicates if ek is the encryption key used by package H

$\texttt{isCrsOf}_{crs}(H) \to B$ indicates if crs is the CRS used by package H

$\texttt{garbage}(\cdot) \to H$ returns a garbage object

$\texttt{eq}(H, H') \to B$ indicates whether H and H' are handles for identical objects

Figure 4.61: Methods offered by the adversary's operation functionality $\mathcal{O}_{adv}$

4.5.3 The Adversary's Operation Module

All methods shown in Figure 4.61 are offered to the adversary by his operation module[30], except $\texttt{decrypt}_{dk}$ which is only offered in the corruption scenarios where the corresponding player is corrupt. Their implementations follow straight-forwardly from the implementation of the players' operation modules, with the exception that they work directly on handles instead of indirectly through references.

[30]The operations given to the adversary are determined by the methods needed by the translator $\mathcal{T}^\star$ in Section 4.5.4, ie. the adversary need not be given more methods than what is needed by $\mathcal{T}^\star$, making its construction the crucial point at which to determine the interface offered to the adversary.

4.5.4 Soundness of the Intermediate Model

As part of the soundness theorem we first show that a real-world environment cannot distinguish between interacting with $\mathcal{RW}(Sys)$ or $\mathcal{I}(Sys)$ for our systems Sys in consideration. To this end we need to introduce the concept of a *translator* $\mathcal{T}$ parameterised by the corruption scenario and making the two interpretations appear similar[31]. Throughout this section we use $\mathcal{T}[\mathcal{I}(Sys)]$ to denote hybrid interpretations where all crypto channels to the environment are rewired to run through $\mathcal{T}$, and plain channels are left untouched (the bitstrings sent on them already use the same format in the two interpretations). We stress that while the simulator in ideal protocols is per-protocol and must be constructed as part of the analysis, the translator introduced here is per-framework and is constructed once and for all.[32].

We first consider the case where $(Sys^{AB}, Sys^{A}, Sys^{B})$ is a well-formed real protocol, but only focus on the first two cases as the third is symmetrical to the second. Our aim is to show that $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}_{\star}^{\mathcal{H}}\left[\mathcal{I}(Sys^{\mathcal{H}})\right]$ for the translator $\mathcal{T}_{\star}$ defined below, but as a first step we show that these equivalences hold for the more powerful translator $\mathcal{T}_{true,real}$. Through a series of translators we then use the indistinguishability properties of the cryptographic primitives to show that $\mathcal{T}_{\star}^{\mathcal{H}}\left[\mathcal{I}(Sys^{\mathcal{H}})\right]$ is indistinguishable from $\mathcal{T}_{true,real}^{\mathcal{H}}\left[\mathcal{I}(Sys^{\mathcal{H}})\right]$, and the result follows.

Let $\mathcal{T}_{true,real}^{AB}$ be the translator defined in Figure 4.62. This translator emulates a setup functionality $\mathcal{F}_{setup}$ running in `ideal` mode and therefore obtains the common reference strings crs_A and crs_B for the two honest players; by computational zero-knowledge of the NIZK scheme the environment cannot distinguish by the fact that the translator is using this mode instead of mode `real`. The translator also emulates the global memory functionality $\mathcal{F}_{mem}$ and may hence look inside its data objects beyond what is allowed by $\mathcal{O}_{adv}$.

Lemma 4.5.2. *We have* $\mathcal{RW}(Sys^{AB}) \overset{c}{\sim} \mathcal{T}_{true,real}^{AB}\left[\mathcal{I}(Sys^{AB})\right]$.

Proof. We proceed by arguing that the two interpretations are indistinguishable at each activation by the environment. Firstly, since the environment may in this corruption scenario only activate the honest entities through plain channels it is immediately clear that the bitstrings sent by the environment may easily be translated to a matching counterpart in the intermediate interpretation.

Secondly, to argue that the honest entities behave the same on each activation we use that the relevant primitives are well-spread and hence the two interpretations with overwhelming probability agree on when two commitments or encryptions are identical; this is needed for the storing and verification methods to agree. By correctness of the encryption scheme it also follows that the two interpretations agree on the plaintext values of encryptions.

Thirdly, by looking inside the global memory the translator may obtain both values and randomness from the handles of the intermediate model that allows it to produce a leakage of commitments and encryptions distributed as in $\mathcal{RW}(Sys^{AB})$; to do this correctly for evaluation packages it needs to keep the ϵ mapping of already processed encryptions. All proofs may be produced using **Prove** since the cryptographic programmes in a real protocol can only produce proofs for true statements. However, since no information is stored in the memory about the randomness to use when generating the proofs we need the τ mapping to store already translated packages. Specifically, consider the case where $\mathcal{T}_{true,real}^{AB}$ must translate handle H

[31] In UC-terms the translator is simply a simulator for $\mathcal{F}_{aux}$ used to show that the real-world interpretation is a realisation of the intermediate interpretation. However, we use this wording to avoid too much overload.

[32] If we instead considered a framework with e.g. symmetric encryption then we would need a new translator. It might be possible to compose several translators without having to redo all proofs, and thereby making it easier to extend the primitives supported by symbolic analysis. In particular, if the protocol class does not allow mixing two sets of primitives then it seems reasonable to assume that these translators would compose to a framework supporting the combined primitives as long as they are used in a mutually exclusive fashion. We do not investigate this further but refer to [CW11] for results in this direction.

- Handle H received on channel $leak_{AB,i}$ (ie. from honest A):
  - $\texttt{isValue}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isPair}(H)$ returns $\texttt{true}$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain bitstring BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.63 with $ck = ck_A$ and $crs = crs_A$ to obtain bitstring BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.64 with $crs = crs_A$ to obtain bitstring BS; send BS on the corresponding channel
  - $\texttt{isEvalPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.65 with $ck = ck_A$ and $crs = crs_A$ to obtain bitstring BS; send it on the corresponding channel
- Handle H received on channel $leak_{BA,j}$ (ie. from honest B):
  - $\texttt{isValue}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isPair}(H)$ returns $\texttt{true}$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain bitstring BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.63 with $ck = ck_B$ and $crs = crs_B$ to obtain bitstring BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.64 with $crs = crs_B$ to obtain bitstring BS; send BS on the corresponding channel
  - $\texttt{isEvalPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.65 with $ck = ck_B$ and $crs = crs_B$ to obtain bitstring BS; send it on the corresponding channel
- Bitstring BS received on channel $infl_{AB,i}$ or $infl_{BA,j}$ (ie. from the adversary):
  - BS match $\langle \texttt{value} : V \rangle$: send BS on the corresponding channel
  - BS match $\langle \texttt{constant} : Cn \rangle$: send BS on the corresponding channel
  - BS match $\langle \texttt{pair} : BS_1, BS_2 \rangle$: recursively process BS_1 and BS_2 to obtain handles H_1 and H_2; let $H = \texttt{pair}(H_1, H_2)$ and send it on the corresponding channel
  - otherwise use $\texttt{garbage}(\cdot)$ to create and send a garbage handle

Figure 4.62: Translator $\mathcal{T}^{AB}_{true,real}$

Processing of handle H with $\texttt{isComPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:
  1. look inside $\texttt{comOf}(H)$ to obtain V and R
  2. compute $D \leftarrow \mathbf{Com}_{ck}(V, R)$ and $\pi_U \leftarrow \mathbf{Prove}_{U,crs}(D, ck, V, R)$
  3. let $BS = [\texttt{comPackage} : D, ck, \pi_U, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.63: Translator $\mathcal{T}^{AB}_{true,real}$ – commitment package from honest player

Processing of handle H with $\texttt{isEncPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise
  1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
  2. let $H_C = \texttt{encOf}(H)$ and look inside H_C to obtain V and R
  3. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$ and $\pi_T \leftarrow \mathbf{Prove}_{T,crs}(C, ek, V, R)$, and store $\epsilon(H_C) \mapsto C$
  4. let $BS = [\texttt{encPackage} : C, ek, \pi_T, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.64: Translator $\mathcal{T}^{AB}_{true,real}$ – encryption package from honest player

Processing handle H with $\texttt{isEvalPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:

1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
2. for $i \in [2]$ look inside $\texttt{comOf}_i(H)$ to obtain W_i, S_i and compute $D_i \leftarrow \mathbf{Com}_{ck}(W_i, S_i)$
3. for $i \in [2]$ let $C_i = \epsilon(\texttt{encOf}_i(H))$ and pick fresh randomness $R \in \{0,1\}^\kappa$
4. compute $C \leftarrow \mathbf{Eval}_{e,ek}(C_1, C_2, W_1, W_2, R)$ and store $\epsilon(\texttt{encOf}(H)) \mapsto C$
5. compute $\pi_e \leftarrow \mathbf{Prove}_{e,crs}(C, C_1, C_2, ek, D_1, D_2, ck, W_1, S_1, W_2, S_2, R)$
6. let $BS = [\texttt{evalPackage} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, store $\pi(H) \mapsto BS$, and return BS

Figure 4.65: Translator $\mathcal{T}^{AB}_{true,real}$ – evaluation package from honest player

presenting a commitment package (Figure 4.63). Assume first that $\tau(H) = \bot$ meaning that this package has not been processed before. By looking inside the data object associated with $\texttt{comOf}(H)$ in the global memory it may obtain a value V and a randomness R, and hence the commitment D generated using **Com** is distributed exactly as in $\mathcal{RW}(Sys^{AB})$. As for the proof, $\tau(H) = \bot$ implies that this is the first time π_U will be send hence it will also be distributed exactly as in $\mathcal{RW}(Sys^{AB})$. For the case where $\tau(H) \neq \bot$ we need to argue that resending the same translation is ok. If we had ignored τ and instead processed the package again, looking in the global memory we would have ended up with the same commitment D, so the only thing that could potentially be different in $\mathcal{RW}(Sys^{AB})$ is the proof. However, since the protocol is well-formed we have that $\texttt{commit}_{U,ck,crs}$ has been invoked at most once for V, R by the sending programme and hence the package sent in $\mathcal{RW}(Sys^{AB})$ also contains the same proof. The cases where H is an encryption package (Figure 4.64) or an evaluation package (Figure 4.65) are similar. □

Next, let $\mathcal{T}^{A}_{true,real}$ be the translator defined in Figure 4.66. This translator also emulates the global memory functionality and the setup functionality $\mathcal{F}_{setup}$ running in `ideal` mode but obtains a simulation trapdoor $simtd_A$ for player A and an extraction trapdoor $extd_B$ for player B; again by the NIZK scheme being computational zero-knowledge and extractable the environment cannot tell that difference from the setup alone.

Lemma 4.5.3. *We have* $\mathcal{RW}(Sys^A) \overset{c}{\sim} \mathcal{T}^{A}_{true,real}[\mathcal{I}(Sys^A)]$.

Proof. The argument goes in much the same way as for when both players are honest, relying on the logic of `storeCrypto` in the honest player's operation module to not only ensure that extraction is possible, but also to reject encryption and evaluation packages that would break identity or which cannot be translated for more subtle reasons.

One thing to note is that when translating commitment packages from the corrupt player we use the commitment D as the randomness component in the translated data object. This is to ensure that the identity of commitments is preserved: we cannot use the extractable R since the binding property of the scheme does not rule out the possibility that the adversary can come up with $R \neq R'$ such that $\mathbf{Com}_{ck}(V, R) = \mathbf{Com}_{ck}(V, R')$. Using D as the randomness component guarantees that identity is preserved amoung all commitments created by the corrupt player, which is enough since the honest player will never compare one of them to a commitment that he himself created (the commitment keys are different).

Likewise, we also use C as the randomness component when translating encryption packages. However this time we cannot rely on the encryption key to separate the encryptions and must instead keep the σ mapping which tags each encryption with type and creator, and ensures that

- Handle H received on channel $leak_{AB,i}$ (ie. from honest A):
  - $\texttt{isValue}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$ returns $\texttt{true}$: send H on the corresponding channel
  - $\texttt{isPair}(H)$ returns $\texttt{true}$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain bitstrings BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.67 with $ck = ck_A$ and $crs = crs_A$ to obtain bitstring BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.68 with $crs = crs_A$ to obtain bitstring BS; send BS on the corresponding channel
  - $\texttt{isEvalPack}(H)$ returns $\texttt{true}$: use the procedure in Figure 4.69 with $ck = ck_A$ and $crs = crs_A$ to obtain bitstring BS; send it on the corresponding channel
- Bitstring BS received on crypto channel $infl_{BA,j}$ (ie. from corrupt B):
  - match BS with $\langle \texttt{value} : V \rangle$: send BS on the corresponding channel
  - match BS with $\langle \texttt{constant} : Cn \rangle$: send BS on the corresponding channel
  - match BS with $\langle \texttt{pair} : BS_1, BS_2 \rangle$: recursively process BS_1 and BS_2 to obtain H_1 and H_2; let $H = \texttt{pair}(H_1, H_2)$ and send it on the corresponding channel
  - match BS with $[\texttt{comPack} : \ldots]$: use the procedure in Figure 4.70 with $ck = ck_B$, $crs = crs_B$, and $extd = extd_B$ to obtain handle H; send it on the corresponding channel
  - match BS with $[\texttt{encPack} : \ldots]$: use the procedure in Figure 4.71 with $crs = crs_B$ and $extd = extd_B$ to obtain handle H; send it on the corresponding channel
  - match BS with $[\texttt{evalPack} : \ldots]$: use the procedure in Figure 4.72 with $ck = ck_B$, $crs = crs_B$, and $extd = extd_B$ to obtain handle H; send it on the corresponding channel
  - otherwise use $\texttt{garbage}(\cdot)$ to create and send a garbage handle

Figure 4.66: Translator $\mathcal{T}^A_{true,real}$

Processing of handle H with $\texttt{isComPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:
  1. look inside $\texttt{comOf}(H)$ to obtain V and R
  2. compute $D \leftarrow \mathbf{Com}_{ck}(V, R)$ and $\pi_U \leftarrow \mathbf{Prove}_{U,crs}(D, ck, V, R)$
  3. let $BS = [\texttt{comPackage} : D, ck, \pi_U, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.67: Translator $\mathcal{T}^A_{true,real}$ – commitment package from honest player

Processing of handle H with $\texttt{isEncPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise
  1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
  2. let $H_C = \texttt{encOf}(H)$ and look inside H_C to obtain V and R
  3. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$ and $\pi_T \leftarrow \mathbf{Prove}_{T,crs}(C, ek, V, R)$
  4. store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{encme}(H_C)$
  5. let $BS = [\texttt{encPackage} : C, ek, \pi_T, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.68: Translator $\mathcal{T}^A_{true,real}$ – encryption package from honest player

Processing handle H with $\texttt{isEvalPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise
  1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
  2. for $i \in [2]$ let $C_i = \epsilon(\texttt{encOf}_i(H))$
  3. for $i \in [2]$ look inside $\texttt{comOf}_i(H)$ to obtain W_i, S_i and compute $D_i \leftarrow \mathbf{Com}_{ck}(W_i, S_i)$
  4. pick fresh randomness $R \in \{0,1\}^\kappa$ and compute $C \leftarrow \mathbf{Eval}_{e,ek}(C_1, C_2, W_1, W_2, R)$
  5. compute $\pi_e \leftarrow \mathbf{Prove}_{e,crs}(C, C_1, C_2, ek, D_1, D_2, ck, W_1, S_1, W_2, S_2, R)$
  6. let $H_C = \texttt{encOf}(H)$, and store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{evalme}(H_C)$
  7. let $BS = [\texttt{evalPackage} : C, C_1, C_2, ek, D_1, D_2, ek, \pi_e, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.69: Translator $\mathcal{T}^A_{true,real}$ – evaluation package from honest player

Processing of bitstring $BS = [\texttt{comPack} : D, ck, \pi_U, crs]$:

- if $\mathbf{Ver}_{U,crs}(D, ck, \pi_U)$ succeeds then:
  1. compute $(V, \cdot) \leftarrow \mathbf{Extract}_{U,extd}(D, ck, \pi_U)$, let $H = \texttt{commit}_{U,ck,crs}\big(V, D\big)$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.70: Translator $\mathcal{T}^A_{true,real}$ – commitment package from corrupt player

Processing of bitstring $BS = [\texttt{encPack} : C, ek, \pi_T, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{T,crs}(C, ek, \pi_T)$ succeeds, and $\sigma(C, ek) \in \{\perp, \texttt{encother}(\cdot)\}$ then:
  1. compute $(V, \cdot) \leftarrow \mathbf{Extract}_{T,extd}(C, ek, \pi_T)$, and let $H = \texttt{encrypt}_{T,ek,crs}\big(V, C\big)$
  2. let $H_C = \texttt{encOf}(H)$, store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{encother}(H_C)$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.71: Translator $\mathcal{T}^A_{true,real}$ – encryption package from corrupt player

Processing of bitstring $BS = [\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{e,crs}(C, \ldots, \pi_e)$ succ., and $\sigma(C, ek) = \texttt{evalother}(BS, H, \cdot)$ then return H
- else if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{e,crs}(C, \ldots, \pi_e)$ succeeds, $\sigma(C, ek) = \perp$, and $\sigma(C_i, ek) \in \{\texttt{encother}(H_{C_i}), \texttt{evalother}(\cdot, \cdot, H_{C_i}), \texttt{encme}(H_{C_i}), \texttt{evalme}(H_{C_i})\}$ for both $i \in [2]$ then:
  1. compute $(W_1, \cdot, W_2, \cdot, \cdot) \leftarrow \mathbf{Extract}_{e,extd}(C, C_1, C_2, ek, D_1, D_2, ck, \pi_e)$ for D_1 and D_2
  2. let $H = \texttt{eval}_{e,ek,ck,crs}\big(H_{C_1}, H_{C_2}, W_1, D_1, W_2, D_2\big)$ and $H_C = \texttt{encOf}(H)$
  3. store $\sigma(C, ek) \mapsto \texttt{evalother}(BS, H, H_C)$ and $\epsilon(H_C) \mapsto C$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a new garbage handle

Figure 4.72: Translator $\mathcal{T}^A_{true,real}$ – evaluation package from corrupt player

- Handle H received on crypto channel $leak_{AB,i}$ (ie. from honest A):
  - $\texttt{isValue}(H)$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$: send H on the corresponding channel
  - $\texttt{isPair}(H)$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$: use the procedure in Figure 4.74 with $ck = ck_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$: use the procedure in Figure 4.75 with $crs = crs_A$ and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEvalPack}(H)$: use the procedure in Figure 4.76 with $ck = ck_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
- Handle H received on crypto channel $leak_{BA,j}$ (ie. from honest B):
  - $\texttt{isValue}(H)$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$: send H on the corresponding channel
  - $\texttt{isPair}(H)$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$: use the procedure in Figure 4.74 with $ck = ck_B$, $crs = crs_B$, and $simtd = simtd_B$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$: use the procedure in Figure 4.75 with $crs = crs_B$ and $simtd = simtd_B$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEvalPack}(H)$: use the procedure in Figure 4.76 with $ck = ck_B$, $crs = crs_B$, and $simtd = simtd_B$ to obtain BS; send it on the corresponding channel
- Bitstring BS received on channel $infl_{AB,i}$ or $infl_{BA,j}$ (ie. from the adversary):
  - $BS = [\texttt{value} : V]$: send BS on the corresponding channel
  - $BS = [\texttt{constant} : Cn]$: send BS on the corresponding channel
  - $BS = [\texttt{pair} : BS_1, BS_2]$: recursively process BS_1 and BS_2 to obtain H_1 and H_2; let $H = [\texttt{pair} : H_1, H_2]$ and send it on the corresponding channel
  - otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.73: Translator $\mathcal{T}_\star^{AB}$

identity just needs to be preserved within each separation[33]: an encryption package is rejected (Figure 4.71) unless its encryption C has never been seen before (case $\sigma(C, ek) = \perp$) in which case identity is trivially preserved, or it has only been seen as part of encryption packages that also came from the corrupt player (case $\sigma(C, ek) = \texttt{encother}(\cdot)$), in which case identity is preserved since C is again used as the randomness component and the encryption scheme is correct so that the extracted value V is the same; likewise, an evaluation package is rejected (Figure 4.72) unless its encryption C has never been seen before (case $\sigma(C, ek) = \perp$), or it has only been seen as part of the exact same evaluation package (case $\sigma(C, ek) = \texttt{evalother}(\cdot)$) in which case identity is preserved since the same handle H is used. Note that σ also ensures that an evaluation package from the corrupt player can actually be translated by rejecting evaluations for which we do not already have a translation of the sub-encryptions. □

[33] Our assumptions on the encryption scheme do not rule out the possibility that the environment can extract both V, R from an honestly generated encryption C if he for instance knows the decryption key. He may then form a valid encryption package with C' and send it back to the honest player. In this case $C' = C$, yet the translator will yield a data object $H_{C'}$ where $H_C \neq H_{C'}$ as the former has R as randomness and the latter C.

Processing of handle H with $\texttt{isComPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:

1. let $D = \delta(\texttt{comOf}(H))$; if $D = \bot$ then
   a) pick randomness $R \in \{0,1\}^\kappa$, compute $D \leftarrow \mathbf{Com}_{ck}(0, R)$, and store $\delta(\texttt{comOf}(H)) \mapsto D$
2. compute $\pi_U \leftarrow \mathbf{SimProve}_{U,simtd}(D, ck)$
3. let $BS = [\texttt{comPackage} : D, ck, \pi_U, crs]$, store $\tau(H) = BS$, and return BS

Figure 4.74: Translator $\mathcal{T}_\star^{AB}$ – commitment package from honest player

Processing of handle H with $\texttt{isEncPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:

1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
2. let $H_C = \texttt{encOf}(H)$ and $C = \epsilon(H_C)$; if $C = \bot$ then
   a) pick randomness $R \in \{0,1\}^\kappa$, compute $C \leftarrow \mathbf{Enc}_{ek}(0, R)$, and store $\epsilon(H_C) \mapsto C$
3. compute $\pi_T \leftarrow \mathbf{SimProve}_{T,simtd}(C, ek)$
4. let $BS = [\texttt{encPackage} : C, ek, \pi_T, crs]$, store $\tau(H) = BS$, and return BS

Figure 4.75: Translator $\mathcal{T}_\star^{AB}$ – encryption package from honest player to honest player

Processing handle H with $\texttt{isEvalPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:

1. use $\texttt{isEkOf}_{ek}(H)$ to determine which encryption key ek to use
2. for $i \in [2]$ let $H_{D_i} = \texttt{comOf}_i(H)$ and $D_i = \delta(H_{D_i})$; if $D_i = \bot$ then
   a) pick randomness $R_i \in \{0,1\}^\kappa$, compute $D_i \leftarrow \mathbf{Com}_{ck}(0, R)$, and update $\delta(H_{D_i}) \mapsto D_1$
3. for $i \in [2]$ let $C_i = \epsilon(\texttt{encOf}_i(H))$ and pick fresh randomness $R \in \{0,1\}^\kappa$
4. compute $C \leftarrow \mathbf{Enc}_{ek}(0, R)$ and $\pi_e \leftarrow \mathbf{SimProve}_{e,simtd}(C, C_1, C_2, ek, D_1, D_2, ek, crs)$
5. let $H_C = \texttt{encOf}(H)$, and store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto H_C$
6. let $BS = [\texttt{evalPackage} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, update $\tau(H) \mapsto BS$, and return BS

Figure 4.76: Translator $\mathcal{T}_\star^{AB}$ – evaluation package from honest player to honest player

Let translator $\mathcal{T}_\star^{AB}$ be given by Figure 4.73 and translator $\mathcal{T}_\star^{A}$ by Figure 4.77. They are very similar to their $\mathcal{T}_{true,real}$ counterpart but have introduced mapping δ for storing already processed commitments from honest players, and have extended the use of mapping ϵ to also contain already processed encryptions from honest players. Intuitively this is possible since all proofs are simulated, and means that the translators no longer needs to look into the data objects in the global memory to obtain the randomness components R. In fact they only use the methods offered by $\mathcal{O}_{adv}$ to the adversary[34].

Lemma 4.5.4. *For any well-formed real protocol* $(Sys^{\mathcal{H}})_{\mathcal{H}}$ *we have* $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}_\star^{\mathcal{H}}\big[\mathcal{I}(Sys^{\mathcal{H}})\big]$.

Proof. Using the above results we proceed to show the result by a series of hybrid interpretations progressively changing $\mathcal{T}_{true,real}^{\mathcal{H}}$ into $\mathcal{T}_\star^{\mathcal{H}}$. Indistinguishability of the hybrids follows from the indistinguishability properties of the underlying cryptographic primitives. Note that throughout we use that well-formedness ensures that the same randomness is never used twice; this is important for indistinguishability of the primitives.

[34] Equality of objects, eq, is needed when the translator looks in its mappings ϵ, δ, and σ: it would not be enough for it to simply compare handles as the global memory allows the same object to be created under different handles.

- Handle H received on crypto channel $leak_{AB,i}$ (ie. from honest A):
  - $\texttt{isValue}(H)$: send H on the corresponding channel
  - $\texttt{isConstant}(H)$: send H on the corresponding channel
  - $\texttt{isPair}(H)$: recursively process $\texttt{first}(H)$ and $\texttt{second}(H)$ to obtain BS_1 and BS_2; let $BS = [\texttt{pair} : BS_1, BS_2]$ and send it on the corresponding channel
  - $\texttt{isComPack}(H)$: use the procedure in Figure 4.78 with $ck = ck_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$ and $\texttt{isEkOf}_{ek_A}(H)$: use the procedure in Figure 4.79 with $ek = ek_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEncPack}(H)$ and $\texttt{isEkOf}_{ek_B}(H)$; use the procedure in Figure 4.80 with $ek = ek_B$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEvalPack}(h)$ and $\texttt{isEkOf}_{ek_A}(H)$: use the procedure in Figure 4.81 with $ek = ek_A$, $ck = ck_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
  - $\texttt{isEvalPack}(h)$ and $\texttt{isEkOf}_{ek_B}(H)$: use the procedure in Figure 4.82 with $ek = ek_B$, $ck = ck_A$, $crs = crs_A$, and $simtd = simtd_A$ to obtain BS; send it on the corresponding channel
- Bitstring BS received on crypto channel $infl_{BA,j}$ (ie. from corrupt B):
  - $BS = [\texttt{value} : V]$: send BS on the corresponding channel
  - $BS = [\texttt{constant} : Cn]$: send BS on the corresponding channel
  - $BS = [\texttt{pair} : BS_1, BS_2]$: recursively process BS_1 and BS_2 to obtain H_1 and H_2; let $H = [\texttt{pair} : H_1, H_2]$ and send it on the corresponding channel
  - $BS = [\texttt{comPack} : \ldots]$: use the procedure in Figure 4.83 with $ck = ck_B$, $crs = crs_B$, and $extd = extd_B$ to obtain H; send it on the corresponding channel
  - $BS = [\texttt{encPack} : \ldots]$: use the procedure in Figure 4.84 with $crs = crs_B$ and $extd = extd_B$ to obtain H; send it on the corresponding channel
  - $BS = [\texttt{evalPack} : \ldots]$: use the procedure in Figure 4.85 with $ck = ck_B$, $crs = crs_B$, and $extd = extd_B$ to obtain H; send it on the corresponding channel
  - otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.77: Translator $\mathcal{T}_\star^A$

Processing of handle H with $\texttt{isComPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise;
  1. let $H_D = \texttt{comOf}(H)$ and $D = \delta(H_D)$; if $D = \perp$ then
     a) pick randomness $R \in \{0,1\}^\kappa$; compute $D \leftarrow \mathbf{Com}_{ck}(0, R)$, and update $\delta(H_D) \mapsto D$
  2. compute $\pi_U \leftarrow \mathbf{Sim}_{U,simtd}(D, ck)$
  3. let $BS = [\texttt{comPackage} : D, ck, \pi_U, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.78: Translator $\mathcal{T}_\star^A$ – commitment package from honest player

Processing of handle H with $\texttt{isEncPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise
  1. let $H_C = \texttt{encOf}(H)$ and $C = \epsilon(H_C)$; if $C = \perp$ then
     a) pick $R \in \{0,1\}^\kappa$, comp. $C \leftarrow \mathbf{Enc}_{ek}(0, R)$, and store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{encme}(H_C)$
  2. compute $\pi_T \leftarrow \mathbf{SimProve}_{T,simtd}(C, ek)$
  3. let $BS = [\texttt{encPackage} : C, ek, \pi_T, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.79: Translator $\mathcal{T}_\star^A$ – encryption package from honest player under honest key

Processing of handle H with $\mathsf{isEncPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise
  1. let $H_C = \texttt{encOf}(H)$ and $C = \epsilon(H_C)$; if $C = \perp$ then
     a) let $V = \texttt{decrypt}_{dk}(H_C)$ and pick fresh randomness $R \in \{0,1\}^\kappa$
     b) compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$, and store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{encme}(H_C)$
  2. compute $\pi_T \leftarrow \mathbf{SimProve}_{T,simtd}(C, ek)$
  3. let $BS = [\texttt{encPackage} : C, ek, \pi_T, crs]$, store $\tau(H) \mapsto BS$, and return BS

Figure 4.80: Translator $\mathcal{T}_\star^A$ – encryption package from honest player under corrupt key

Processing handle H with $\mathsf{isEvalPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:
  1. for $i \in [2]$ let $H_{D_i} = \texttt{comOf}_i(H)$ and $D_i = \delta(H_{D_i})$; if $D_i = \perp$ then
     a) pick randomness $R_i \in \{0,1\}^\kappa$, compute $D_i \leftarrow \mathbf{Com}_{ck}(0, R)$, and update $\delta(H_{D_i}) \mapsto D_1$
  2. for $i \in [2]$ let $C_i = \epsilon(\texttt{encOf}_i(H))$
  3. pick fresh randomness $R \in \{0,1\}^\kappa$ and let $H_C = \texttt{encOf}(H)$
  4. compute $C \leftarrow \mathbf{Enc}_{ek}(0, R)$ and $\pi_e \leftarrow \mathbf{SimProve}_{e,simtd}(C, C_1, C_2, ek, D_1, D_2, ck)$
  5. store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{evalme}(H_C)$
  6. let $BS = [\texttt{evalPackage} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, store $\tau(H) = BS$, and return BS

Figure 4.81: Translator $\mathcal{T}_\star^A$ – evaluation package from honest player under honest key

Processing of handle H with $\mathsf{isEvalPack}(H)$:

- if $\tau(H) = BS$ then return BS; otherwise:
  1. for $i \in [2]$ let $H_{D_i} = \texttt{comOf}_i(H)$ and $D_i = \delta(H_{D_i})$; if $D_i = \perp$ then
     a) pick randomness $R_i \in \{0,1\}^\kappa$, compute $D_i \leftarrow \mathbf{Com}_{ck}(0, R)$, and update $\delta(H_{D_i}) \mapsto D_1$
  2. for $i \in [2]$ let $C_i = \epsilon(\texttt{encOf}_i(H))$
  3. pick fresh randomness $R \in \{0,1\}^\kappa$, and let $H_C = \texttt{encOf}(H)$ and $V = \texttt{decrypt}_{dk}(H_C)$
  4. compute $C \leftarrow \mathbf{Enc}_{ek}(V, R)$ and $\pi_e \leftarrow \mathbf{SimProve}_{e,simtd}(C, C_1, C_2, ek, D_1, D_2, ck)$
  5. store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{evalme}(H_C)$
  6. let $BS = [\texttt{evalPackage} : C, C_1, ek, D_1, D_2, ck, \pi_e, crs]$, store $\tau(H) \mapsto BS$, return BS

Figure 4.82: Translator $\mathcal{T}_\star^A$ – evaluation package from honest player under corrupt key

Processing of bitstring $BS = [\texttt{comPack} : D, ck, \pi_U, crs]$:

- if $\mathbf{Ver}_U(D, ck, \pi_U, crs)$ succeeds then:
  1. compute $(V, \cdot) \leftarrow \mathbf{Extract}_{U,extd}(D, ck, \pi_U)$, let $H = \texttt{commit}_{U,ck,crs}\big(V, D\big)$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.83: Translator $\mathcal{T}_\star^A$ – commitment package from corrupt player

Processing of bitstring $BS = [\texttt{encPack} : C, ek, \pi_T, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{T,crs}(C, ek, \pi_T)$ succeeds, and $\sigma(C, ek) \in \{\perp, \texttt{encother}(\cdot)\}$ then:
  1. compute $(V, \cdot) \leftarrow \mathbf{Extract}_{T,extd}(C, ek, \pi_T)$, and let $H = \texttt{encrypt}_{T,ek,crs}\big(V, C\big)$
  2. let $H_C = \texttt{encOf}(H)$, store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \texttt{encother}(H_C)$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a garbage handle

Figure 4.84: Translator $\mathcal{T}_\star^A$ – encryption package from corrupt player

Processing of bitstring $BS = [\texttt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{e,crs}(C, \ldots, \pi_e)$ succ., and $\sigma(C, ek) = \texttt{evalother}(BS, H, \cdot)$ then return H
- else if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{e,crs}(C, \ldots, \pi_e)$ succeeds, $\sigma(C, ek) = \bot$, and $\sigma(C_i, ek) \in \{\texttt{encother}(H_{C_i}), \texttt{evalother}(\cdot, \cdot, H_{C_i}), \texttt{encme}(H_{C_i}), \texttt{evalme}(H_{C_i})\}$ for both $i \in [2]$ then:
  1. compute $(W_1, \cdot, W_2, \cdot, \cdot) \leftarrow \mathbf{Extract}_{e,extd}(C, C_1, C_2, ek, D_1, D_2, ck, \pi_e)$ for D_1 and D_2
  2. let $H = \texttt{eval}_{e,ek,ck,crs}(H_{C_1}, H_{C_2}, W_1, D_1, W_2, D_2)$ and $H_C = \texttt{encOf}(H)$
  3. store $\sigma(C, ek) \mapsto \texttt{evalother}(BS, H, H_C)$ and $\epsilon(H_C) \mapsto C$, and return H
- otherwise use $\texttt{garbage}(\cdot)$ to create and return a new garbage handle

Figure 4.85: Translator $\mathcal{T}^A_\star$ – evaluation package from corrupt player

- Let n be the number of proofs sent by honest players in the execution. In interpretation $\mathcal{T}^{\mathcal{H}}_{sim,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ for $0 \leq i \leq n$ we use the simulation trapdoors to create the first i proofs from honest players using **SimProve** instead of **Prove**. Indistinguishability between $\mathcal{T}^{\mathcal{H}}_{sim,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ and $\mathcal{T}^{\mathcal{H}}_{sim,i+1}[\mathcal{I}(Sys^{\mathcal{H}})]$ follows by the NIZK scheme being computational zero-knowledge on true statements. Since n is polynomial in κ we get indistinguishability through-out the entire series.
- In interpretation $\mathcal{T}^{\mathcal{H}}_{rand}[\mathcal{I}(Sys^{\mathcal{H}})]$ we ignore the randomness symbols supplied by the honest players and let the translator chose fresh randomness on its own instead. This is possible because the protocol is well-formed and all proofs are simulated, yet it requires us to maintain the mapping δ for commitments and ϵ for encryptions.
- Let n be the number of commitments sent by honest players in the execution. In interpretations $\mathcal{T}^{\mathcal{H}}_{com,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ for $0 \leq i \leq n$ we replace the values in the first i commitments from honest players with constants instead of the actual values. Indistinguishability between $\mathcal{T}^{\mathcal{H}}_{com,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ and $\mathcal{T}^{\mathcal{H}}_{com,i+1}[\mathcal{I}(Sys^{\mathcal{H}})]$ follows by the commitment scheme being computationally hiding. Since n is polynomial in κ we then get indistinguishability through-out the entire series.
- Let n be the number of evaluation packages sent by honest players in the execution. In interpretations $\mathcal{T}^{\mathcal{H}}_{eval,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ for $0 \leq i \leq n$ we produce the ciphertext C in the first i evaluation packages from honest players using **Enc** instead of using **Eval**. Indistinguishability between $\mathcal{T}^{\mathcal{H}}_{eval,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ and $\mathcal{T}^{\mathcal{H}}_{eval,i+1}[\mathcal{I}(Sys^{\mathcal{H}})]$ follows by the encryption scheme being history hiding. Since n is polynomial in κ we then get indistinguishability through-out the entire series.
- Let n be the number of encryptions sent by honest players *to honest players* in the execution. In interpretations $\mathcal{T}^{\mathcal{H}}_{enc,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ for $0 \leq i \leq n$ we replace the values in the first i of these encryptions with constants instead of the actual values; for encryptions for corrupt player we keep the actual values. Indistinguishability between $\mathcal{T}^{\mathcal{H}}_{enc,i}[\mathcal{I}(Sys^{\mathcal{H}})]$ and $\mathcal{T}^{\mathcal{H}}_{enc,i+1}[\mathcal{I}(Sys^{\mathcal{H}})]$ follows by IND-CPA of the encryption scheme. Since n is polynomial in κ we then get indistinguishability through-out the entire series.
- Finally, in interpretation $\mathcal{T}^{\mathcal{H}}_{dec}[\mathcal{I}(Sys^{\mathcal{H}})]$ the global memory functionality $\mathcal{F}_{mem}$ is moved outside so that the translator now only has access to it through $\mathcal{O}_{adv}$. However, at this point the only situations where $\mathcal{T}^{\mathcal{H}}_{dec}$ needs to look inside the memory is to obtain the correct value for encryptions sent to a corrupt player, and this can be done using the `decrypt` method instead.

In summary we get that $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}^{\mathcal{H}}_{dec}[\mathcal{I}(Sys^{\mathcal{H}})]$, and the result follows since $\mathcal{T}^{\mathcal{H}}_{dec} = \mathcal{T}^{\mathcal{H}}_{\star}$. □

To get a similar result for an ideal protocol we start with a hybrid interpretation $\mathcal{T}^{\mathcal{H}}_{true,ideal}$ similar to $\mathcal{T}^{\mathcal{H}}_{true,real}$ but parameterised by the programmes in the system; this is needed for the translator to know which $\mathsf{simeval}_e$ operation was used to create each evaluation package (and in turn whether **Eval** or **Enc** was used) as it cannot decide this from the received handles and interacting with $\mathcal{O}_{adv}$ alone. Since all proofs are already simulated we skip the first step of going from $\mathcal{T}^{\mathcal{H}}_{true,real}$ to $\mathcal{T}^{\mathcal{H}}_{\star}$ but otherwise apply the remaining sequence of hybrids as in Lemma 4.5.4.

Lemma 4.5.5. *For any well-formed ideal protocol* $(Sys^{\mathcal{H}})_{\mathcal{H}}$ *we have* $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}^{\mathcal{H}}_{\star}\left[\mathcal{I}(Sys^{\mathcal{H}})\right]$.

We can then state the soundness result.

Theorem 4.5.6 (Soundness of Intermediate Interpretation)**.** *Let* $Sys^{\mathcal{H}}_1$ *and* $Sys^{\mathcal{H}}_2$ *be two well-formed systems. If* $\mathcal{I}(Sys^{\mathcal{H}}_1) \overset{c}{\sim} \mathcal{I}(Sys^{\mathcal{H}}_2)$ *then* $\mathcal{RW}(Sys^{\mathcal{H}}_1) \overset{c}{\sim} \mathcal{RW}(Sys^{\mathcal{H}}_2)$.

Proof. By assumption no polynomially bounded ITM $\mathcal{Z}'$ can tell the difference between $\mathcal{I}(Sys^{\mathcal{H}}_1)$ and $\mathcal{I}(Sys^{\mathcal{H}}_2)$ while having access only to $\mathcal{O}_{adv}$, in particular no $\mathcal{Z}' = \mathcal{Z} \diamond \mathcal{T}^{\mathcal{H}}_{\star}$ for a polynomially bounded ITM $\mathcal{Z}$. The result then follows by Lemma 4.5.4 and 4.5.5. □

Corollary 4.5.7. *Let* $(Sys^{\mathcal{H}}_{real})_{\mathcal{H}}$ *be a real protocol* ϕ *and let* $(Sys^{\mathcal{H}}_{ideal})_{\mathcal{H}}$ *be an ideal protocol with target functionality* $\mathcal{F}$*. If* $\mathcal{I}(Sys^{\mathcal{H}}_{real}) \overset{c}{\sim} \mathcal{I}(Sys^{\mathcal{H}}_{ideal})$ *for all three* $\mathcal{H} \in \{AB, A, B\}$ *then* ϕ *realises* $M_{\mathcal{F}}$ *(with inlined operation module) under static corruption.*

Proof. We first note that by combining from the setup functionality with the M_P machines for simulator(s), authenticated channels, and simulated functionalities, we obtain an syntactically correct UC-simulator *CombSim* for $M_{\mathcal{F}}$. We then use the assumption and Theorem 4.5.6. □

4.6 Symbolic Model and Interpretation

We now give a symbolic model and interpretation tailored to be a conservative approximation of the intermediate model. We use the dialect in [BAF05] of the applied-pi calculus [AF01] as the underlying framework since this provides us with an unified way of expressing honest entities, the powers of the adversary[35], and indistinguishability. Moreover, we will later use that automated verification tools exist for this calculus in the form of ProVerif [Bla04, BAF05].

4.6.1 Symbolic Model

For the symbolic model we assume a modelling **Vals** of the values in the domain, i.e. for each integer in the domain in consideration there is a unique abstract term[36] ranged over by v. Likewise we also assume a modelling of all constants in $\mathbf{Consts} \cup \{\mathbf{true}, \mathbf{false}, \mathbf{garbage}\}$. Let $\mathcal{N}$ (also called *names*) be a countable set of atomic symbols used to model randomness r, secret key material $dk, extd$, and ports p. A term t is then build from names n in $\mathcal{N}$, a countable set of variables $x, y, z, \ldots$, and the constructor symbols in Figure 4.86. The $\mathbf{proof}_{(\cdot)}$ constructors are unavailable to the adversary.

pair	for pairings
ek, **crs**	for keys
com, **enc**	for commitments and encryptions
$\mathbf{proof}_U$, **comPack**	for commitment packages
$\mathbf{proof}_T$, **encPack**	for encryption packages
$\mathbf{proof}_e$, **evalPack**	for evaluation packages

Figure 4.86: Term constructor symbols

The destructor symbols are given in Figure 4.87, and we also use t to range over terms with destructors. Only the $\mathbf{eval}_e$ destructor is unavailable to the adversary. The reason for this is that in order to keep the symbolic model suitable for automated analysis, we do not wish to symbolically model the composition of randomness from encryptions when performing homomorphic evaluations. On the other hand, the evaluated encryption cannot use randomness supplied by the adversary nor the randomness of only one of the input encryptions, as both cases would allow the adversary to easily guess the plaintext. Our solution is then to use fresh unknown randomness. However, we cannot express this directly in the equational theories suitable for ProVerif[37]; instead the private $\mathbf{eval}_e$ destructor takes a randomness r as input and we give the adversary access to it only through an oracle process (more below).

Processes Q are built from the grammar described in Figure 4.88, where t is a term, u is a name or port, p is a port, and x is a variable. The nil process does nothing and represents a halted state. The new $u; Q$ process is used for name and port restriction. Intuitively, the let $x = t$ in Q else Q' process tries to evaluate t to t' by reducing it using the equational theory and the rewrite rules (over which the calculus is parameterised); if it is successful it binds it to x in Q and proceeds as this process; if it fails (because there is no matching rewrite rule for

[35] As usual, these are given by his deductive powers (ie. his ability to form new messages from old ones) and his testing powers (ie. his ability to distinguish messages). The private function symbols allowed by this dialect of the calculus allows us to better express the adversary's exact powers.

[36] One may for instance obtain such a model by having an atomic term for each value. Alternatively one could have constructors used to represent numbers in unary or binary.

[37] And if we could, we couldn't reveal it to the adversary either, as this would allow him to deduce too much, in turn making it difficult for him to prove that he correctly formed an evaluation package.

$\mathbf{isComPack}, \mathbf{verComPack}_U$	for commitments
$\mathbf{isEncPack}, \mathbf{verEncPack}_T$	for encryptions
$\mathbf{eval}_e, \mathbf{isEvalPack}, \mathbf{verEvalPack}_e$	for evaluations
$\mathbf{dec}, \mathbf{extractCom}, \mathbf{extractEnc}$	for decryption
$\mathbf{extractEval}_1, \mathbf{extractEval}_2$	and extraction
$\mathbf{ckOf}, \mathbf{ekOf}, \mathbf{crsOf},$	
$\mathbf{comOf}, \mathbf{comOf}_1, \mathbf{comOf}_2,$	for packages
$\mathbf{encOf}, \mathbf{encOf}_1, \mathbf{encOf}_2$	
$\mathbf{isValue}, \mathbf{eqValue}, \mathbf{inType}_U, \mathbf{inType}_T$	for values
$\mathbf{isConst}, \mathbf{eqConst}_c$	for constants
$\mathbf{isPair}, \mathbf{first}, \mathbf{second}$	for pairings
$\mathbf{equals}$	for identity checking

Figure 4.87: Term destructor symbols

nil	$\text{in}[p, x]; Q$	let $x = t$ in Q else Q'	$Q \mid\mid Q'$
new $u; Q$	$\text{out}[p, t]; Q$	if $t = t'$ then Q else Q'	$!\, Q$

Figure 4.88: Process syntax

a destructor) then it proceeds as Q' instead. When Q' is clear from the context we shall also write let $x = t;\ Q$. The if $t = t'$ then Q else Q' process is just syntactic sugar[38] but intuitively proceeds as Q if t and t' can be rewritten to terms equivalent according to the equtional theory, and as Q' if not. Again we will at times omit the "else Q'" part when Q' is clear from the context. Finally, $Q \mid\mid Q'$ denotes parallel composition, and $!\, Q$ unbounded replication.

Let an *evaluation context* $\mathcal{E}$ be a process with a hole, built from $[\_]$, $\mathcal{E} \mid\mid Q$, $Q \mid\mid \mathcal{E}$ and new $n; \mathcal{E}$. We obtain $\mathcal{E}[Q]$ as the result of filling the hole in $\mathcal{E}$ with Q. We say that a process Q is *closed* if all its variables are bound through an input or a let construction. We may now capture the operational semantics of processes by two relations, namely *structural equivalence* and *reduction*. Structural equivalence, denoted by $\equiv$, is the smallest equivalence relation on processes that is closed under application of evaluation contexts and standard rules such as associativity and commutativity of the parallel operator (see [AF01, BAF05] for details). Reduction, denoted by $\rightarrow$, is the smallest relation closed under structural equivalence, application of evaluation contexts, and rules:

$$!\, Q \rightarrow Q \mid\mid !\, Q$$

$$\text{out}[p, t]; Q_1 \mid\mid \text{in}[p, x]; Q_2 \rightarrow Q_1 \mid\mid Q_2\{t/x\}$$

$$\text{let } x = t \text{ in } Q \text{ else } Q' \rightarrow \begin{cases} Q\{t'/x\} & \text{when } t \Downarrow t' \text{ for some } t' \\ Q' & \text{otherwise} \end{cases}$$

where $t \Downarrow t'$ indicates that t may be rewritten to t' containing no destructors using the rewrite rules and the equational theory. Our rewrite rules are given in Figure 4.89 and we only need a

[38] Namely, if $t = t'$ then Q else Q' is defined as usual as let $x = \mathbf{equals}(t, t')$ in Q else Q' for x free in Q.

trivial equational theory[39]. We write $\rightarrow^*$ for the reflexive and transitive closure of reduction.

Our equivalence notion for formalising *symbolic indistinguishability* is observational equivalence as defined in [AF01]. Here we write $Q{\downarrow_p}$ when Q can send an observable message on port p; that is, when $Q \rightarrow^* \mathcal{E}[\text{out}[p,t];Q']$ for some term t, some process Q', and some evaluation context $\mathcal{E}$ that does not bind p.

Definition 4.6.1 (Symbolic indistinguishability)**.** Symbolic indistinguishability*, denoted* $\overset{s}{\sim}$*, is the largest symmetric relation* $\mathcal{R}$ *on closed processes* Q_1 *and* Q_2 *such that* $Q_1 \mathcal{R} Q_2$ *implies:*

1. *if* $Q_1{\downarrow_p}$ *then* $Q_2{\downarrow_p}$
2. *if* $Q_1 \rightarrow Q_1'$ *then there exists* Q_2' *such that* $Q_2 \rightarrow^* Q_2'$ *and* $Q_1' \mathcal{R} Q_2'$
3. $\mathcal{E}[Q_1] \mathcal{R} \mathcal{E}[Q_2]$ *for all evaluation contexts* $\mathcal{E}$

Intuitively, a context may represent an attacker, and two processes are symbolic indistinguishable if they cannot be distinguished by any attacker at any step: every output step in an execution of process Q_1 must have an indistinguishable equivalent output step in the execution of process Q_2, and vice versa; if not then there exists a context that "breaks" the equivalence.

Note however that the definition uses an existential quantification: if $Q_1 \overset{s}{\sim} Q_2$ then we only know that a reduction of Q_1 can be matched by *some* reduction by Q_2. Since we allow private connections in our protocols this has implication for the soundness result in Section 4.6.5 as symbolic indistinguishability only guarantees that *some* scheduling of two systems make them indistinguishable; for soundness we need that this holds for the (token-based) scheduling used in the computational model.

4.6.2 Programme Interpretation

Using the model from above we may now give a symbolic interpretation of a programme P in the form of a process Q_P with private access to an implementation of the operations available to it[40]. Invocation of one of these operations is done by sending a message on a dedicated port, say $p^{call}_{\mathsf{commit}}$, and receiving the result back on a corresponding $p^{ret}_{\mathsf{commit}}$. Boolean operations such as $\mathsf{isEncPack}$ always return either **true** or **false**, and non-boolean operations return a term unless they abort (say, a check fails) in which case no message is sent back causing Q_P to block. Abusing the notation slightly we shall often write invocations inlined, ie. let $y = \mathsf{encrypt}_{T,ek,crs}(v,r);Q$ instead of $\text{out}[p^{call}_{\mathsf{encrypt}\ldots}, \mathbf{pair}(v,r)];\text{in}[p^{ret}_{\mathsf{encrypt}\ldots}, y];Q$.

Unlike the other models, the symbolic implementation of the operation modules does not include storing methods. Since the checks performed by these are still required by soundness, we assume that the programme itself contains enough instructions such that whenever a message is received, the programmes rejects it if the intermediate interpretation would have done so, as determined by methods `acceptPlain` and `acceptCrypto`. This is easily satisfied for protocols where all messages have a predefined structure, and we have chosen so for simplicity, in particular to avoid encoding the recursive checking of pairings and the σ list of previously received encryptions; avoiding these encodings is desirable from an automated verification point of view.

We introduce a bit of syntactic sugar before giving the interpretation. Let $Q_{p_1}, \ldots, Q_{p_n}$ be processes. We then use the standard trick of writing

$$\text{in}[p_1, x_1];Q_{p_1} + \text{in}[p_2, x_2];Q_{p_2} + \cdots + \text{in}[p_n, x_n];Q_{p_n}$$

[39] The ProVerif manual [Bla11] advocates the use of rewrite rules over equations for efficiency reasons.

[40] As for the computational models this strong separation between the programme and its module is artificial, and in practise the module is simply inlined in the process.

$$\mathbf{isValue}(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in \mathbf{Dom} \qquad \mathbf{isPair}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow \mathbf{true}$$
$$\mathbf{eqValue}(v, v) \rightsquigarrow \mathbf{true} \text{ for all } v \in \mathbf{Dom} \qquad \mathbf{first}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow x_1$$
$$\mathbf{inType}_U(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in U \qquad \mathbf{second}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow x_2$$
$$\mathbf{inType}_T(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in T \qquad \mathbf{eqConst}_c(c) \rightsquigarrow \mathbf{true} \text{ for all } c \in \mathbf{Const}$$
$$\mathbf{isConst}(c) \rightsquigarrow \mathbf{true} \text{ for all } c \in \mathbf{Const}$$

$$\mathbf{peval}_f(v_1, v_2, v_3, v_4) \rightsquigarrow v \text{ for all } v_i \in \mathbf{Dom} \text{ and } v = f(v_1, v_2, v_3, v_4)$$

$$\mathbf{isComPack}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verComPack}_U(\mathbf{comPack}(x_d, x_{ck}, \mathbf{proof}_U(x_d, x_{ck}, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$

$$\mathbf{isEncPack}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verEncPack}_T(\mathbf{encPack}(x_c, x_{ek}, \mathbf{proof}_T(x_c, x_{ek}, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$

$$\mathbf{isEvalPack}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verEvalPack}_e(\mathbf{evalPack}(x_c, x_{c_1}, \ldots, x_{ck}, \mathbf{proof}_e(x_c, x_{c_1}, \ldots, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{eval}_e(\mathbf{enc}(v_1, x_{r_1}, \mathbf{ek}(x_{dk})), \mathbf{enc}(v_2, x_{r_2}, \mathbf{ek}(x_{dk})), v_3, v_4, x_r) \rightsquigarrow \mathbf{enc}(v, x_r, \mathbf{ek}(dk))$$
$$\text{for all } v_i \in \mathbf{Dom} \text{ and } v = \mathbf{peval}_e(v_1, v_2, v_3, v_4)$$
$$\mathbf{dec}(\mathbf{enc}(v, x_r, \mathbf{ek}(x_{dk})), x_{dk}) \rightsquigarrow v$$
$$\mathbf{extractCom}(\mathbf{proof}_U(\mathbf{com}(v, x_r, x_{ck}), x_{ck}, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v$$
$$\mathbf{extractEnc}(\mathbf{proof}_T(\mathbf{enc}(v, x_r, x_{ek}), x_{ek}, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v$$
$$\mathbf{extractEval}_i(\mathbf{proof}_e(\ldots, \mathbf{com}(v_1, x_{r_1}, x_{ck}), \mathbf{com}(v_2, x_{r_2}, x_{ck}), \ldots, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v_i \text{ for } i \in \{1, 2\}$$

$$\mathbf{comOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_d$$
$$\mathbf{ckOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ck}$$
$$\mathbf{proofOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{encOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_c$$
$$\mathbf{ekOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_{ek}$$
$$\mathbf{proofOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{encOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_c$$
$$\mathbf{encOf}_i(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{c_i} \text{ for } i \in \{1, 2\}$$
$$\mathbf{ekOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ek}$$
$$\mathbf{comOf}_i(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{d_i} \text{ for } i \in \{1, 2\}$$
$$\mathbf{ckOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ck}$$
$$\mathbf{proofOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{equals}(x, x) \rightsquigarrow \mathbf{true}$$

Figure 4.89: Term rewrite rules

instead of

$$\text{new } p; \left(\begin{array}{l} \text{out}[p, \textbf{token}]; \text{nil} \\ \quad || \ \text{in}[p, x]; \text{in}[p_1, x_1]; Q_{p_1} \\ \quad || \ \text{in}[p, x]; \text{in}[p_2, x_2]; Q_{p_2} \\ \quad || \ \ldots \\ \quad || \ \text{in}[p, x]; \text{in}[p_n, x_n]; Q_{p_n} \end{array} \right)$$

for a process $\sum_i \text{in}[p_i, x_i]; Q_{p_i}$ listening on several ports at once but prevented from responding to more than one of them (p and x are free in all Q_{p_i}). Informally, only one of the processes may receive **token** and hence continue.

To give an interpretation of a programme we start by interpreting the leaves as nil processes. We then iteratively work our way backwards through the edges: consider a programme point Σ with n outgoing edges pointing to $\Sigma_1, \ldots, \Sigma_n$ that have already been interpreted as processes $Q_1, \ldots, Q_n$. We now partition these edges by the port they are listening on. For each partition p_i we may then interpret (sub-)programme P_{p_i} from Σ by an input statement followed by an series of if-then-else processes encoding the conditions, a series of let processes encoding the commands sequence, and ending in an optional output statement. This gives processes $\text{in}[p_i, x_i]; Q_{p_i}$ that may then finally be combined to obtain $Q = \sum_i \text{in}[p_i, x_i]; Q_{p_i}$

As an example consider a node Σ with two input-output edges (to Σ_1 and Σ_2) and one input-only edge (to Σ_3) that all listen on the same port p_{in}, and with respective conditions

$$\begin{aligned} \psi_1 &= \mathsf{isEncPack}(x_{in}) \wedge \mathsf{verEncPack}_{T,ek,crs}(x_{in}) \\ \psi_2 &= \mathsf{isEvalPack}(x_{in}) \wedge \mathsf{verEvalPack}_{e,ek,ck,crs}(x_{in}) \\ \psi_3 &= \neg\psi_1 \vee \neg\psi_2 \end{aligned}$$

and respective command sequences

$$\left\{ \frac{\mathsf{decrypt}_{dk}(x_{in})}{y} \right\} \qquad \left\{ \frac{\mathsf{decrypt}_{dk}(x_{in})}{y} \right\} \qquad \emptyset$$

which is well-formed since $\mathsf{isEncPack}(x)$ implies $\neg\mathsf{isEvalPack}(x)$, and vice versa. For the symbolical interpretation we obtain the sequential process Q in Figure 4.90.

4.6.3 Symbolic Implementation of Operation Modules

To form a symbolic operation module for an honest entity we first give a process Q_{op} for each available operation op. We have omitted the simpler operations and only give these processes for commitment, encryption, and evaluation operations in Figure 4.91, 4.92, and 4.93 respectively, where we have also omitted some "else" branches. Note that they follow the intermediate implementation in Section 4.5.2 closely. For a process Q_P for programme P with access to operations $op_1, \ldots, op_n$ we may then form its operation process Q_{box_P} as

$$Q_{box_P} \doteq !\, Q_{op_1} \ || \ \cdots \ || \ !\, Q_{op_n}$$

which we note may be parameterised by keys if P is cryptographic. Since this process is private to Q_P we will have to link them through a series of port restrictions along the lines of

$$\text{new } p^{call}_{op_1}; \text{new } p^{ret}_{op_1}; \ldots; \text{new } p^{call}_{op_n}; \text{new } p^{ret}_{op_n}; \left(Q_P \ || \ Q_{box_P}\right)$$

such that only Q_P may interact with Q_{box_P}.

As for the operations offered to the adversary (see Section 4.5.3) we first note that his operations may all be modelled as above. However, it is also sound to give the symbolic

$$
\begin{array}{l}
Q \doteq \mathrm{in}[p_{in}, x_{in}]; \\
\quad \text{if } \mathsf{isEncPack}(x_{in}) = \mathbf{true} \text{ then} \\
\quad\quad \text{if } \mathsf{verEncPack}_{T,ek,crs}(x_{in}) = \mathbf{true} \text{ then} \\
\quad\quad\quad \text{let } y = \mathsf{decrypt}_{dk}(x_{in}); \\
\quad\quad\quad \mathrm{out}[p_1, x_1]; Q_1 \\
\quad\quad \text{else } Q_3 \\
\quad \text{else if } \mathsf{isEvalPack}(x_{in}) = \mathbf{true} \text{ then} \\
\quad\quad \text{if } \mathsf{verEvalPack}_{e,ek,ck,crs}(x_{in}) = \mathbf{true} \text{ then} \\
\quad\quad\quad \text{let } y = \mathsf{decrypt}_{dk}(x_{in}); \\
\quad\quad\quad \mathrm{out}[p_2, x_2]; Q_2 \\
\quad\quad \text{else } Q_3 \\
\quad \text{else } Q_3
\end{array}
$$

Figure 4.90: Example symbolic interpretation of programme point Σ

$$
\begin{array}{rl}
Q_{\mathsf{verComPack}_{U,ck,crs}} \doteq & \mathrm{in}[p^{call}_{\mathsf{verComPack}_{U,ck,crs}}, x_d]; \\
& \text{if } \mathbf{verComPack}_U(x_d) = \mathbf{true} \text{ then} \\
& \text{if } \mathbf{ckOf}(x_d) = ck \text{ then} \\
& \text{if } \mathbf{crsOf}(x_d) = crs \text{ then} \\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verComPack}_{U,ck,crs}}, \mathbf{true}] \\
\\
Q_{\mathsf{commit}_{U,ck,crs}} \doteq & \mathrm{in}[p^{call}_{\mathsf{commit}_{U,ck,crs}}, (x_v, x_r)]; \\
& \text{if } \mathbf{inType}_U(x_v) = \mathbf{true} \text{ then} \\
& \quad \text{let } x_d = \mathbf{com}(x_v, x_r, ck); \\
& \quad \text{let } x_\pi = \mathbf{proof}_U(x_d, ck, crs); \\
& \quad \mathrm{out}[p^{ret}_{\mathsf{commit}_{U,ck,crs}}, \mathbf{comPack}(x_d, ck, x_\pi, crs)] \\
\\
Q_{\mathsf{simcommit}_{U,ck,simtd}} \doteq & \mathrm{in}[p^{call}_{\mathsf{simcommit}_{U,ck,simtd}}, (x_v, x_r)]; \\
& \text{if } \mathbf{isValue}(x_v) = \mathbf{true} \text{ then} \\
& \quad \text{let } x_d = \mathbf{com}(x_v, x_r, ck); \\
& \quad \text{let } x_\pi = \mathbf{proof}_U(x_d, ck, crs); \\
& \quad \mathrm{out}[p^{ret}_{\mathsf{simcommit}_{U,ck,crs}}, \mathbf{comPack}(x_d, ck, x_\pi, crs)]
\end{array}
$$

Figure 4.91: Symbolic implementation of operations for commitment packages

$$
\begin{aligned}
Q_{\mathsf{verEncPack}_{T,ek,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{verEncPack}_{T,ek,crs}}, x_c];\\
& \text{if } \mathbf{verEncPack}_T(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{ekOf}(x_c) = ek \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verEncPack}_{T,ek,crs}}, \mathbf{true}]\\
\\
Q_{\mathsf{encrypt}_{T,ek,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{encrypt}_{T,ek,crs}}, (x_v, x_r)];\\
& \text{if } \mathbf{inType}_T(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_c = \mathbf{enc}(x_v, x_r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_T(x_c, ek, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{encrypt}_{T,ek,crs}}, \mathbf{encPack}(x_c, ek, x_\pi, crs)]\\
\\
Q_{\mathsf{simencrypt}_{T,ek,simtd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{simencrypt}_{T,ek,crs}}, (x_v, x_r)];\\
& \text{if } \mathbf{isValue}(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_c = \mathbf{enc}(x_v, x_r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_T(x_c, ek, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{simencrypt}_{T,ek,crs}}, \mathbf{encPack}(x_c, ek, x_\pi, crs)]
\end{aligned}
$$

Figure 4.92: Symbolic implementation of operations for encryption packages

adversary more powers than the intermediate adversary, yet it may simplify the analysis. He may use any constructor and destructor as he pleases, except for $\mathbf{eval}_e$, $\mathbf{prove}_U$, $\mathbf{prove}_T$ and $\mathbf{prove}_e$ which we have to grant him explicit access to. To do this we give him access to process Q^{adv}_{box} defined as follows[41]

$$Q^{adv}_{box} \doteq !\, Q^{adv}_{\mathsf{commit}_U} \mid\mid !\, Q^{adv}_{\mathsf{encrypt}_T} \mid\mid !\, Q^{adv}_{\mathsf{eval}_e}$$

using the processes in Figure 4.95.

4.6.4 Symbolic Interpretation

Given a system *Sys* in corruption scenario $\mathcal{H}$ we may use the encoding of programmes from above to form a composed process $Q^{\mathcal{H}}_{honest}$ of all programmes in *Sys* along with their operation modules. By combining this with a process $Q^{\mathcal{H}}_{adv}$ containing Q^{adv}_{box} as well as a process leaking the public and corrupted decryption keys, we obtain our symbolic interpretation:

Definition 4.6.2 (Symbolic Interpretation)**.** *The* symbolic interpretation $\mathcal{S}(Sys)$ *of a well-formed system Sys is given by process* $\mathcal{E}^{\mathcal{H}}_{setup}[Q^{\mathcal{H}}_{honest} \mid\mid Q^{\mathcal{H}}_{adv}]$ *where the setup contexts* $\mathcal{E}^{\mathcal{H}}_{setup}$ *are given in Figure 4.96.*

4.6.5 Soundness of Symbolic Interpretation

Since the symbolic model already matches the intermediate model quite closely, the main issue for the soundness theorem is to ensure that the two notions of equivalence coincide. This in

[41] Note that there is no need to give different boxes in the different corruption scenarios.

$$
\begin{aligned}
Q_{\mathsf{verEvalPack}_{e,ek,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{verEvalPack}_{e,ek,ck,crs}}, (x_c, x_{c_1}, x_{c_2}, x_{d_1}, x_{d_2})];\\
& \text{if } \mathbf{verEvalPack}_e(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{encOf}_i(x_c) = \mathbf{encOf}(c_i) \text{ then}\\
& \text{if } \mathbf{comOf}_i(x_c) = \mathbf{comOf}(d_i) \text{ then}\\
& \text{if } \mathbf{ekOf}(x_c) = ek \text{ then}\\
& \text{if } \mathbf{ckOf}(x_c) = ck \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verEvalPack}_{e,ek,ck,crs}}, \mathbf{true}]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{eval}_{e,ek,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{eval}_{e,ek,ck,crs}}, (x_{c_1}, x_{c_2}, x_{v_1}, x_{r_1}, x_{v_2}, x_{r_2})];\\
& \text{if } \mathbf{ekOf}(x_{c_i}) = ek \text{ then}\\
& \text{if } \mathbf{isValue}(x_{v_i}) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_{c'_i} = \mathbf{encOf}(x_{c_i});\\
& \quad \text{let } x_{d'_i} = \mathbf{com}(x_{v_i}, x_{r_i}, ck);\\
& \quad \text{new } r;\\
& \quad \text{let } x_{c'} = \mathbf{eval}_e(x_{c'_1}, x_{c'_2}, x_{v_1}, x_{v_2}, r);\\
& \quad \text{let } x_\pi = \mathbf{proof}_e(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, crs);\\
& \quad \text{let } x_c = \mathbf{evalPack}(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, x_\pi, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{eval}_{e,ek,ck,crs}}, x_c]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{simeval}_{e,ek,ck,simtd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{eval}_{e,ek,ck,crs}}, (x_v, x_{c_1}, x_{c_2}, x_{d_1}, x_{d_2})];\\
& \text{if } \mathbf{ekOf}(x_{c_i}) = ek \text{ then}\\
& \text{if } \mathbf{ckOf}(x_{d_i}) = ck \text{ then}\\
& \quad \text{let } x_{c'_i} = \mathbf{encOf}(x_{c_i});\\
& \quad \text{let } x_{d'_i} = \mathbf{comOf}(x_{d_i});\\
& \quad \text{new } r;\\
& \quad \text{let } x_{c'} = \mathbf{enc}(x_v, r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_e(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, crs);\\
& \quad \text{let } x_c = \mathbf{evalPack}(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, x_\pi, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{eval}_{e,ek,ck,crs}}, x_c]
\end{aligned}
$$

Figure 4.93: Symbolic implementation of operations for evaluation packages

$$
\begin{aligned}
Q_{\mathsf{decrypt}_{dk}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{decrypt}_{dk}}, x_c];\\
& \text{if } \mathbf{ekOf}(x_c) = ek \text{ then}\\
& \quad \text{let } x_{c'} = \mathbf{encOf}(x_c);\\
& \quad \text{let } x_v = \mathbf{dec}(x_{c'}, dk);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{decrypt}_{dk}}, x_v]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{extractCom}_{extd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{extractCom}_{crs}}, x_d];\\
& \text{if } \mathbf{isComPack}(x_d) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{crsOf}(x_d) = crs \text{ then}\\
& \quad \text{let } x_\pi = \mathbf{proofOf}(x_d);\\
& \quad \text{let } x_v = \mathbf{extractCom}(x_\pi, extd);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{extractCom}_{crs}}, x_v]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{extractEnc}_{extd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{extractEnc}_{crs}}, x_c];\\
& \text{if } \mathbf{isEncPack}(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \text{let } x_\pi = \mathbf{proofOf}(x_c);\\
& \quad \text{let } x_v = \mathbf{extractEnc}(x_\pi, extd);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{extractEnc}_{crs}}, x_v]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{extractEval}_{1,extd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{extractEval}_{1,crs}}, x_c];\\
& \text{if } \mathbf{isEvalPack}(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \text{let } x_\pi = \mathbf{proofOf}(x_c);\\
& \quad \text{let } x_v = \mathbf{extractEval}_1(x_\pi, extd);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{extractEval}_{1,crs}}, x_v]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{extractEval}_{2,extd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{extractEval}_{2,crs}}, x_c];\\
& \text{if } \mathbf{isEvalPack}(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \text{let } x_\pi = \mathbf{proofOf}(x_c);\\
& \quad \text{let } x_v = \mathbf{extractEval}_2(x_\pi, extd);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{extractEval}_{2,crs}}, x_v]
\end{aligned}
$$

Figure 4.94: Symbolic implementation of operations for decryption and extraction

$$
\begin{aligned}
Q^{adv}_{\mathsf{commit}_U} \doteq\ & \mathrm{in}[p^{advcall}_{\mathsf{commit}_U}, (x_v, x_r, x_{ck}, x_{crs})];\\
& \text{if } \mathbf{inType}_U(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_d = \mathbf{com}(x_v, x_r, x_{ck});\\
& \quad \text{let } x_\pi = \mathbf{proof}_U(x_d, x_{ck}, x_{crs});\\
& \quad \mathrm{out}[p^{advret}_{\mathsf{commit}_U}, \mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})]
\end{aligned}
$$

$$
\begin{aligned}
Q^{adv}_{\mathsf{encrypt}_T} \doteq\ & \mathrm{in}[p^{advcall}_{\mathsf{encrypt}_T}, (x_v, x_r, x_{ek}, x_{crs})];\\
& \text{if } \mathbf{inType}_T(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_c = \mathbf{encrypt}(x_v, x_r, x_{ek});\\
& \quad \text{let } x_\pi = \mathbf{proof}_T(x_c, x_{ek}, x_{crs});\\
& \quad \mathrm{out}[p^{advret}_{\mathsf{encrypt}_T}, \mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})]
\end{aligned}
$$

$$
\begin{aligned}
Q^{adv}_{\mathsf{eval}_e} \doteq\ & \mathrm{in}[p^{advcall}_{\mathsf{eval}_e}, (x_{c_1}, x_{c_2}, x_{v_1}, x_{r_1}, x_{v_2}, x_{r_2}, x_{ek}, x_{ck}, x_{crs})];\\
& \text{new } r;\\
& \text{let } x_c = \mathbf{eval}_e(x_{c_1}, x_{c_2}, x_{v_1}, x_{v_2}, r);\\
& \text{let } x_{d_1} = \mathbf{com}(x_{v_1}, x_{r_1}, x_{ck});\\
& \text{let } x_{d_2} = \mathbf{com}(x_{v_2}, x_{r_2}, x_{ck});\\
& \text{let } x_\pi = \mathbf{proof}_e(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_{crs});\\
& \mathrm{out}[p^{advret}_{\mathsf{eval}_e}, \mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})]
\end{aligned}
$$

Figure 4.95: Symbolic implementation of operations for adversary's operations

$$
\begin{aligned}
\mathcal{E}^{AB}_{setup} \doteq\ & \text{new } ck_A, ck_B;\\
& \text{new } dk_A, dk_B;\\
& \text{let } ek_A = \mathbf{ek}(dk_A);\\
& \text{let } ek_B = \mathbf{ek}(dk_B);\\
& \text{new } crs_A, crs_B;\\
& [\_]
\end{aligned}
\qquad
\begin{aligned}
\mathcal{E}^{A}_{setup} \doteq\ & \text{new } ck_A, ck_B;\\
& \text{new } dk_A, dk_B;\\
& \text{let } ek_A = \mathbf{ek}(dk_A);\\
& \text{let } ek_B = \mathbf{ek}(dk_B);\\
& \text{new } crs_A, extd_B;\\
& \text{let } crs_B = \mathbf{crs}(extd_B);\\
& [\_]
\end{aligned}
\qquad
\begin{aligned}
\mathcal{E}^{B}_{setup} \doteq\ & \text{new } ck_A, ck_B;\\
& \text{new } dk_A, dk_B;\\
& \text{let } ek_A = \mathbf{ek}(dk_A);\\
& \text{let } ek_B = \mathbf{ek}(dk_B);\\
& \text{new } extd_A, crs_B;\\
& \text{let } crs_A = \mathbf{crs}(extd_A);\\
& [\_]
\end{aligned}
$$

Figure 4.96: Setup evaluation contexts

turn essentially boils down to ensuring that the scheduling that leads to symbolic equivalence coincides with the scheduling policy used in the computational interpretations[42].

Our solution is to restrict systems such that they allow only one choice of symbolic scheduling, namely that of the computational model. In particular, we have as an invariant that if a system is activated with an input then there is only one execution path to either an output or a deadlock state[43]. Initially this invariant holds because of the protocol model. To ensure that it is preserved during execution it is enough to require that no message is lost, ie. for any strategy of the adversary, if a programme sends a message on a port then the receiving programme is at a programme point where it is listening on that port. The motivation behind this choice is that the two models disagree on what happens when the receiver is not ready: in the computational model the message is lost (read but ignored by the receiver) while in the symbolic model the message hangs around (possibly blocking) until the receiver is ready; this may then lead to non-determinism and several scheduling choices.

Theorem 4.6.3. *Let Sys_1 and Sys_2 be two well-formed systems that do not allow messages to be lost. If $\mathcal{S}(Sys_1) \overset{s}{\sim} \mathcal{S}(Sys_2)$ then $\mathcal{I}(Sys_1) \overset{c}{\sim} \mathcal{I}(Sys_2)$.*

Proof. Let $Q_i = \mathcal{S}(Sys_i)$ and let $\mathcal{Z}$ be any polynomial time environment[44] interacting with either $N_1 = \mathcal{I}(Sys_1)$ or $N_2 = \mathcal{I}(Sys_2)$. If we can show that for all fixed choices of random tape for $\mathcal{Z}$ there is only a negligible probability (over the random tapes of the honest machines) of distinguishing N_1 from N_2 then this implies that they are also indistinguishable when the random tape of $\mathcal{Z}$ is drawn from a distribution instead.

Now assume that the random tape of $\mathcal{Z}$ has been fixed so its first activation becomes deterministic. If its action is an output guess $g \in \{0, 1\}$ then we are done since it would clearly do the same in both cases. Else, if it is an activation of an honest machine in N_i (ie. an invocation of its operation module or sending a message to a programme machine) then we need to argue that when $\mathcal{Z}$ is re-activated it only has negligible probability of distinguishing. To do this we first show that there exists an evaluation context $\mathcal{E}^0$ that when applied to Q_i with overwhelming probability will match the reaction of N_i. Since $Q_1 \overset{s}{\sim} Q_2$ implies $\mathcal{E}^0[Q_1] \overset{s}{\sim} \mathcal{E}^0[Q_2]$ this will then allow us to make conclusions about the reaction of N_1 and N_2.

More concretely, by inspecting the bitstring sent by $\mathcal{Z}$ we may use the mappings on values and constants to show the existence of an evaluation context $\mathcal{E}^0$ that extensionally behaves the same: name restriction is used for the randomness sent to its operation module, **garbage** for the handles (since this is the first activation and hence nothing has been received from the honest machines yet, any handle from the adversary must be a guess that we assume is going to fail), and **pair** for constructing pairings. For each randomness sent we also record its associated name by $\rho(R) \mapsto r$. Below we then show that with overwhelming probability the activation of a machine in N_i ends with a message M_i^0 being sent back to $\mathcal{Z}$ if and only if $\mathcal{E}^0[Q_i]$ evaluates to an output of term t_i^0 on an open port; moreover, M_i^0 and t_i^0 have the same structure. Then, since $\mathcal{E}^0[Q_1] \overset{s}{\sim} \mathcal{E}^0[Q_2]$ as mentioned above we must have that t_1^0 is output if and only if t_2^0 is, and by the operations available to the symbolic adversary they must also have the same structure (otherwise the operations could be used to distinguish the two terms). But this in turn means

[42] This issue arises as a combined consequence of the existential quantification in observation equivalence and the use of private ports. Concretely, we may construct two systems which are indistinguishable in the symbolic mode but trivially distinguishable in the computational model because of the different scheduling policy.

[43] Note that the symbolic adversary may choose to activate a system with more than one input at a time because of the inherited concurrency of the symbolic model. This is not a problem since we only want to show soundness in one direction.

[44] The UC framework gives a precisely notion of polynomial time (in the security parameter κ) for ITMs. What we require here is that the messages send by the environment contain at most polynomially many random bitstrings, and that it only invokes its operation module and the honest programme machines a polynomial number of times; we put no restrictions on the amount of computation that goes into producing the messages.

that with overwhelming probability the only difference between M_1^0 and M_2^0 is their random bitstrings; and since the operation modules always refresh these during `retrieve`, the two have the same distribution from the point of view of $\mathcal{Z}$. But this means that with overwhelming probability, when $\mathcal{Z}$ is re-activated it is done so by a message that is distributed the same in the two cases and hence it cannot distinguish.

To continue the argument for the second activation we use the same approach as before, and show that for all choices of random bitstrings in the message it may only distinguish with negligible probability. Concretely, let M^0 be any message from the distribution of M_1^0 and M_2^0, and consider the (now deterministic) activation of $\mathcal{Z}$ on this message. Again, if its action is an output guess then we are done. Otherwise, if it is an activation of an honest machine then we show by construction there exists an evaluation context that extensionally behaves the same. Unlike what we did before we first need to decompose M^0 in order to correctly interpret the action: having chosen a fresh variable name x_0, we then store in η each handle encountered in M^0 together with a term of **first** and **second** describing its path relative to x_0, ie. if $M^0 = \langle \texttt{pair} : H_1, H_2 \rangle$ then $\eta(H_1) \mapsto \mathbf{first}(x_0)$ and $\eta(H_2) \mapsto \mathbf{second}(x_0)$ afterwards. Similar to before, we may then construct context $\mathcal{E}^1$ that first behaves as $\mathcal{E}^0$, next inputs for x_0, then use name restriction for new randomness, and finally build its output in accordance with the bitstring sent by $\mathcal{Z}$ and the recordings in ρ and η. Again we apply that with overwhelming probability the first activation of N_i by $\mathcal{Z}$ will be matched by $\mathcal{E}^1[Q_i]$, and the same argument can now be applied to show that the second activation will also be matched with overwhelming probability. Furthermore, $\mathcal{E}^1[Q_1] \overset{s}{\sim} \mathcal{E}^1[Q_2]$.

We may continue this approach for the entire execution and by our assumption that there are at most polynomially many activations we obtain the desired result.

Finally, we need to argue that there is only a negligible probability of a mismatch between N_i and Q_i for each activation. Since we have assumed that no message is lost we know that a priori the two interpretations agree on the sequence of programmes activated as no non-determinism arises[45] in the symbolic execution. This means that the only point where the sequences may diverge is if a clash between the randomly chosen bitstrings of length κ occurs, either because an honest machine chose the same by coincidence or because the environment managed to "guess" one[46]. However, since each activation of N_i compares and generates at most polynomially many of these, the probability that a clash occurs is negligible; note that here we need that the bitstrings sent by $\mathcal{Z}$ may only contain polynomially many random bitstrings. □

[45] We of course also use the conditions are mutually exclusive and that each port only has one receiver. This means that the only point where non-determinism may occur is if an activation allowed a "stuck" output process to finally react with an input process in the symbolic interpretation; but by the assumption that the systems do not allow messages to be lost no output process can get stuck in the first place.

[46] A clash cannot happen in the symbolic model, not least because the adversary is incapable of such guessing. Concretely, there does not exist an evaluation context matching a successful guess (an unsuccessful guess is interpreted as either a fresh name or **garbage** depending on type).

4.7 Analysis of OT Protocol in ProVerif

In this section we illustrate how the ProVerif tool may be used in proving the OT protocol of [DNO08] secure. After fixing the domain we massage the processes from the symbolic interpretation to fit with ProVerif; to keep with the idea of automated analysis this step is done in a somewhat systematic way, although no algorithm is given. We then successfully verify the protocol with ProVerif, and as a sanity check show that the tool correctly discovers expected attacks on intentionally flawed versions of the protocol.

4.7.1 Instantiating The Model

We fix the domain to $\{0, 1, 2\}$ and use atomic symbols **zero**, **one**, and **two** to encode these values; this allows us to hardcode the arithmetic of $\mathbf{peval}_f$ and in turn also $\mathbf{eval}_e$. The types are $dom = \{\mathbf{zero}, \mathbf{one}, \mathbf{two}\}$ and $bit = \{\mathbf{zero}, \mathbf{one}\}$, and by inspecting the protocol we see that we need constants $\{\mathbf{getInput}, \mathbf{bReceived}, \mathbf{xsReceived}, \mathbf{finish}, \mathbf{deliver}\}$ besides the default $\{\mathbf{true}, \mathbf{false}, \mathbf{garbage}\}$.

From the symbolic interpretation we obtain (inlined) processes for each of the programmes in the two protocols. As an example the processes Q_S, Q_R for the two players programmes are shown in Figure 4.97, and the processes for the ideal functionality and simulators when both players honest are shown in Figure 4.98. The process for an authenticated channel is simply

$$Q_{Auth_{AB}} \doteq \text{in}[send_{AB}, x];\ \text{out}[leak_{AB}, x];\ \text{in}[infl_{AB}, \mathbf{deliver}];\ \text{out}[receive_{AB}, x];\ \text{nil}$$

taking a single input, leaking it, and delivering it when told to by the environment.

$Q_S \doteq$ new r, r_0, r_1;
in$[receive_{RS}, c_b]$;
if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
if $\mathbf{ekOf}(c_b) = ek_R$ then
if $\mathbf{crsOf}(c_b) = crs_R$ then
out$[out^S_{OT}, \mathbf{getInput}]$;
in$[in^S_{OT}, x_{01}]$;
if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
let $x_0 = \mathbf{first}(x_{01})$;
let $x_1 = \mathbf{second}(x_{01})$;
if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
let $c'_b = \mathbf{encOf}(c_b)$;
let $c'_x = \mathbf{eval}_{sel}(c'_b, x_0, x_1, r)$;
let $d_0 = \mathbf{com}(x_0, r_0, ck_S)$;
let $d_1 = \mathbf{com}(x_1, r_1, ck_S)$;
let $p_x = \mathbf{proof}_{sel}(c'_x, c'_b, ek_R, d_0, d_1, ck_S, crs_S)$;
let $c_x = \mathbf{evalPack}(c'_x, c'_b, \ldots, ck_S, p_x, crs_S)$;
out$[send_{SR}, c_x]$

$Q_R \doteq$ new r_b;
in$[in^R_{OT}, b]$;
if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
let $c'_b = \mathbf{enc}(b, r_b, ek_R)$;
let $p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
out$[send_{RS}, c_b]$;
in$[receive_{SR}, c_x]$;
if $\mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true}$ then
if $\mathbf{encOf}_1(c_x) = c_b$ then
if $\mathbf{ekOf}(c_x) = ek_R$ then
if $\mathbf{crsOf}(c_b) = crs_R$ then
let $x_b = \mathbf{dec}(\mathbf{encOf}(c_x), dk_R)$;
out$[out^R_{OT}, x_b]$

Figure 4.97: Process Q_S for sender (left) and process Q_R for receiver (right)

$Q_{\mathcal{F}_{OT}^{SR}} \doteq$ in$[in_{OT}^{R}, b]$;
if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
out$[leak_{OT}^{R}, \mathbf{bReceived}]$;
in$[infl_{OT}^{S}, \mathbf{getInput}]$;
out$[out_{OT}^{S}, \mathbf{getInput}]$;
in$[in_{OT}^{S}, x_{01}]$;
if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
let $x_0 = \mathbf{first}(x_{01})$;
let $x_1 = \mathbf{second}(x_{01})$;
if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
out$[leak_{OT}^{S}, \mathbf{xsReceived}]$;
in$[infl_{OT}^{R}, \mathbf{finish}]$;
if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
out$[out_{OT}^{R}, x_0]$
else
out$[out_{OT}^{R}, x_1]$

$Q_{Sim_{OT}^{SR,S}} \doteq$ new r, r_0, r_1;
in$[receive_{RS}, c_b]$;
if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
if $\mathbf{ekOf}(c_b) = ek_R$ then
if $\mathbf{crsOf}(c_b) = crs_R$ then
out$[infl_{OT}^{S}, \mathbf{getInput}]$;
in$[leak_{OT}^{S}, \mathbf{xsReceived}]$;
let $c'_b = \mathbf{encOf}(c_b)$;
let $c'_x = \mathbf{eval}_e(c'_b, \mathbf{zero}, \mathbf{zero}, r)$;
let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S)$;
let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S)$;
let $p_x = \mathbf{proof}_{sel}(c'_x, c'_b, \ldots, crs_S)$;
let $c'_x = \mathbf{evalPack}_{sel}(c'_x, c'_b, \ldots, p_x, crs_S)$;
out$[send_{SR}, c'_x]$

$Q_{Sim_{OT}^{SR,R}} \doteq$ new r_b;
in$[leak_{OT}^{R}, \mathbf{bReceived}]$;
let $c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R)$;
let $p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
out$[send_{RS}, c_b]$;
in$[receive_{SR}, c_x]$;
if $\mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true}$ then
if $\mathbf{encOf}_1(c_x) = c'_b$ then
if $\mathbf{ekOf}(c_x) = ek_R$ then
if $\mathbf{ckOf}(c_x) = ck_S$ then
if $\mathbf{crsOf}(c_x) = crs_S$ then
out$[infl_{OT}^{R}, \mathbf{finish}]$

Figure 4.98: Process for ideal functionality $\mathcal{F}_{OT}^{SR}$ (left) and simulators $Sim_{OT}^{SR,S}$ (right, top) and $Sim_{OT}^{SR,R}$ (right, bottom) for when both players are honest

4.7.2 Massaging Processes for ProVerif

Recall that we want to check the following three equivalences:

$$\mathcal{E}^{SR}[Q_S \mid\mid Q_{Auth_{SR}} \mid\mid Q_{Auth_{RS}} \mid\mid Q_R] \overset{s}{\sim} \mathcal{E}^{SR}[Q_{\mathcal{F}^{SR}} \mid\mid Q_{Sim^{SR,S}} \mid\mid Q_{Auth_{SR}} \mid\mid Q_{Auth_{RS}} \mid\mid Q_{Sim^{SR,R}}]$$
$$\mathcal{E}^{S}[Q_S] \overset{s}{\sim} \mathcal{E}^{S}[Q_{\mathcal{F}^{S}} \mid\mid Q_{Sim^{S}}] \qquad \mathcal{E}^{R}[Q_R] \overset{s}{\sim} \mathcal{E}^{R}[Q_{\mathcal{F}^{R}} \mid\mid Q_{Sim^{R}}]$$

where the evaluation contexts $\mathcal{E}^{SR}$, $\mathcal{E}^{S}$, and $\mathcal{E}^{R}$ take care of setting up keys, restricting ports, and giving leakage and Q_{abox} to the adversary. However, we need to massage the processes before feeding them to ProVerif. More precisely, the tool does not check symbolic equivalence directly but instead checks a strictly stronger *diff equivalence* that requires the two processes Q_1, Q_2 in question to be given by a single *biprocess* B that may be projected to give respectively $Q_1 = \mathit{left}(B)$ and $Q_2 = \mathit{right}(B)$. To specify a biprocess we add a term construction

$$\text{choice}[t_{\mathit{left}} \cdot t_{\mathit{right}}]$$

that intuitively collapses to t_{left} in $\mathit{left}(B)$, and t_{right} in $\mathit{right}(B)$. Note that this implies that processes Q_1, Q_2 must have the same structure and only differ on terms. ProVerif will then check if these two are diff equivalent (see [BAF05] for details).

Consider first the case where both players are honest. The following procedure[47] first gets rid of the obvious structural differences by merging the processes on each side of the equation into as few new processes as possible. In the case of the OT protocol it turns out that a single process is enough on both sides since both protocols are "sequential" in the sense that whenever the protocol expects an input from the environment there is only one open input port as explained next. In the case of the real protocol the processes are initially

$$\text{in}[\mathit{receive}_{RS}, c_b]; Q_0 \;\mid\mid\; \text{in}[\mathit{send}_{SR}, x]; Q_1 \;\mid\mid\; \text{in}[\mathit{send}_{RS}, x]; Q_2 \;\mid\mid\; \text{in}[\mathit{in}^R_{OT}, b]; Q_3$$

for some processes Q_i and where the only open input port is in^R_{OT}. An input on in^R_{OT} will then result in an output on open port leak_{RS} and the processes

$$\text{in}[\mathit{receive}_{RS}, c_b]; Q_0 \;\mid\mid\; \text{in}[\mathit{send}_{SB}, x]; Q_1 \;\mid\mid\; \text{in}[\mathit{infl}_{RS}, x]; Q'_2 \;\mid\mid\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

representing the next state of protocol. This in turn leads to

$$\text{in}[\mathit{in}^S_{OT}, x_{01}]; Q'_0 \;\mid\mid\; \text{in}[\mathit{send}_{SB}, x]; Q_1 \;\mid\mid\; \text{nil} \;\mid\mid\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

followed by

$$\text{nil} \;\mid\mid\; \text{in}[\mathit{infl}_{SB}, x]; Q'_1 \;\mid\mid\; \text{nil} \;\mid\mid\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

and finally

$$\text{nil} \;\mid\mid\; \text{nil} \;\mid\mid\; \text{nil} \;\mid\mid\; \text{nil}$$

where still only one input port is open each time. At each of these *protocol points* we may represent the further behaviour of the protocol by a single process for each of the open input port[48]; this process just performs the concatenated checks and method invocations of all processes activated until there is an output on an open port. Doing so for the real protocol we obtain the single process in the left part of Figure 4.99. For the ideal protocol we obtain the process in the left part of Figure 4.100.

[47] For readability we here present the procedure as working on processes instead of on programmes. An implementation could work on the programme trees instead.

[48] Note that if there are more than one open input port at a protocol point then we need more than one process to represent the further behaviour in the general case. However, in the special case where there are several open input ports yet all but one of them immediately leads to a deadlock we may still use just one process (in fact, one simple programme).

$\text{in}[in^{R}_{OT}, b];$
$\quad$ if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
$\quad$ new r_b;
$\quad$ let $c'_b = \mathbf{enc}(b, r_b, ek_R)$;
$\quad$ let $p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
$\quad$ let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\quad$ $\text{out}[leak_{RS}, c_b]$;
$\text{in}[infl_{RS}, \mathbf{deliver}]$;
$\quad$ if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
$\quad$ if $\mathbf{ekOf}(c_b) = ek_R$ then
$\quad$ if $\mathbf{crsOf}(c_b) = crs_R$ then
$\quad$ $\text{out}[out^{S}_{OT}, \mathbf{getInput}]$;
$\text{in}[in^{S}_{OT}, x_{01}]$;
$\quad$ if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
$\quad$ let $x_0 = \mathbf{first}(x_{01})$;
$\quad$ let $x_1 = \mathbf{second}(x_{01})$;
$\quad$ if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
$\quad$ if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
$\quad$ new r, r_0, r_1;
$\quad$ let $c'_b = \mathbf{encOf}(c_b)$;
$\quad$ let $c'_x = \mathbf{eval}_{sel}(c'_b, x_0, x_1, r)$;
$\quad$ let $d_0 = \mathbf{com}(x_0, r_0, ck_S)$;
$\quad$ let $d_1 = \mathbf{com}(x_1, r_1, ck_S)$;
$\quad$ let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S)$;
$\quad$ let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
$\quad$ $\text{out}[leak_{SR}, c_x]$;
$\text{in}[infl_{SR}, \mathbf{deliver}]$;
$\quad$ if $\mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true}$ then
$\quad$ if $\mathbf{encOf}_1(c_x) = c'_b$ then
$\quad$ if $\mathbf{ekOf}(c_x) = ek_R$ then
$\quad$ if $\mathbf{ckOf}(c_x) = ck_S$ then
$\quad$ if $\mathbf{crsOf}(c_b) = crs_S$ then
$\quad$ let $x_b = \mathbf{dec}(\mathbf{encOf}(c_x), dk_R)$;
$\quad$ $\text{out}[out^{R}_{OT}, x_b]$

$Q^{SR}_{left} \doteq \text{in}[in^{R}_{OT}, b];$
$\quad$ if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
$\quad$ new r_b;
$\quad$ let $c'_b = \mathbf{enc}(b, r_b, ek_R)$;
$\quad$ let $p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
$\quad$ let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\quad$ $\text{out}[leak_{RS}, c_b]$;
$\text{in}[in^{S}_{OT}, x_{01}]$;
$\quad$ if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
$\quad$ let $x_0 = \mathbf{first}(x_{01})$;
$\quad$ let $x_1 = \mathbf{second}(x_{01})$;
$\quad$ if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
$\quad$ if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
$\quad$ new r, r_0, r_1;
$\quad$ let $c'_x = \mathbf{eval}_{sel}(c'_b, x_0, x_1, r)$;
$\quad$ let $d_0 = \mathbf{com}(x_0, r_0, ck_S)$;
$\quad$ let $d_1 = \mathbf{com}(x_1, r_1, ck_S)$;
$\quad$ let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S)$;
$\quad$ let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
$\quad$ $\text{out}[leak_{SR}, c_x]$;
$\text{in}[infl_{SR}, \mathbf{deliver}]$;
$\quad$ let $x_b = \mathbf{dec}(c'_x, dk_R)$;
$\quad$ $\text{out}[out^{R}_{OT}, x_b]$

Figure 4.99: The merged processes from the real protocol, with the naive concatenation on the left and the simplified Q^{SR}_{left} on the right

in$[in^R_{OT}, b]$;
  if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
  new r_b;
  let $c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R)$;
  let $p_b = \mathbf{proof}_{bit}(c_b, ek_R, crs_R)$;
  let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
  out$[leak_{RS}, c_b]$;
in$[infl_{RS}, \mathbf{deliver}]$
  if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
  if $\mathbf{ekOf}(c_b) = ek_R$ then
  if $\mathbf{crsOf}(c_b) = crs_R$ then
  out$[out^S_{OT}, \mathbf{getInput}]$;
in$[in^S_{OT}, x_{01}]$;
  if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
  let $x_0 = \mathbf{first}(x_{01})$;
  let $x_1 = \mathbf{second}(x_{01})$;
  if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
  if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
  new r, r_0, r_1;
  let $c'_b = \mathbf{encOf}(c_b)$;
  let $c'_x = \mathbf{eval}_e(c'_b, \mathbf{zero}, \mathbf{zero}, r)$;
  let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S)$;
  let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S)$;
  let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S)$;
  let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
  out$[leak_{SR}, c_x]$;
in$[infl_{SR}, \mathbf{deliver}]$;
  if $\mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true}$ then
  if $\mathbf{encOf}_1(c_x) = c'_b$ then
  if $\mathbf{ekOf}(c_x) = ek_R$ then
  if $\mathbf{ckOf}(c_x) = ck_S$ then
  if $\mathbf{crsOf}(c_x) = crs_S$ then
  if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
    out$[out^R_{OT}, x_0]$
  else
    out$[out^R_{OT}, x_1]$

$Q^{SR}_{right} \doteq$ in$[in^R_{OT}, b]$;
  if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
  new r_b;
  let $c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R)$;
  let $p_b = \mathbf{proof}_{bit}(c_b, ek_R, crs_R)$;
  let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
  out$[leak_{RS}, c_b]$;
in$[in^S_{OT}, x_{01}]$;
  if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
  let $x_0 = \mathbf{first}(x_{01})$;
  let $x_1 = \mathbf{second}(x_{01})$;
  if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
  if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
  new r, r_0, r_1;
  let $c'_x = \mathbf{eval}_e(c'_b, \mathbf{zero}, \mathbf{zero}, r)$;
  let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S)$;
  let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S)$;
  let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S)$;
  let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
  out$[leak_{SR}, c_x]$;
in$[infl_{SR}, \mathbf{deliver}]$;
  if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
    out$[out^R_{OT}, x_0]$
  else
    out$[out^R_{OT}, x_1]$

Figure 4.100: The merged processes from the ideal protocol, with the naive concatenation on the left and the simplified Q^{SR}_{right} on the right

Although it would now be possible to attempt a merger between the two processes to form a biprocess, this may be made easier by first removing trivial operations.

Consider again the process for the real protocol in the left part of Figure 4.99. By the definition of c_b we see that the three checks if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then, if $\mathbf{ekOf}(c_b) = ek_R$ then, and if $\mathbf{crsOf}(c_b) = crs_R$ then will always be satisfied, and it is hence sound to remove them[49]. This leaves an input on open port $infl_{RS}$ followed immediately by an output on open port out^S_{OT}; removing this is also sound. Continuing with these transformations we obtain the process Q^{SR}_{left} in the right part of Figure 4.99, and a similar reasoning allows us to soundly simplify the ideal protocol to process Q^{SR}_{right} in the right part of Figure 4.100. To finally form the biprocess B^{SR}_{right} for when both players are honest we notice that the only place where Q^{SR}_{left} and Q^{SR}_{right} differ by more than terms are at the final step: the real protocol performs a decryption of c_b while the ideal protocol tests the value of b. Adding a definition of x_b in Q^{SR}_{right} is sound, and so is matching $\text{out}[out^R_{OT}, x_b]$ against both branches of the test in Q^{SR}_{right}. Doing this we obtain the biprocess shown in Figure 4.101.

When only S is honest we may likewise concatenate and simplify the relevant processes to obtain the two new processes Q^S_{left} and Q^S_{right} given in Figure 4.102 that may be merged to form biprocess B^S in Figure 4.103. The same holds for when only R is honest; in this case Q^R_{left} and Q^R_{right} from Figure 4.104 yields biprocess B^R shown in Figure 4.105.

Note that the procedure has preserved equivalence between the two processes in the sense that if the resulting two processes are equivalent then so are the initial two. An important point here is that since destructors may fail when reduced, the defining let statement for a term t with destructors cannot be moved around arbitrarily: we must first ensure that there are enough checks so that t cannot fail, or ensure that t is evaluated in exactly the same activations as it was originally. Similarly, when copying a let statement for a term t from one process to another as part of forming the biprocesses, we must ensure that if t is not copied into a choice construct then no destructors in t can fail; this is for instance the case when forming B^S_{OT} since the let statement for b is copied to the left part but must be outside a choice construct due to the nature of the diff equivalence (ProVerif will yield a false negative in this case).

4.7.3 Automating The Analysis Using ProVerif

Although we may feed the three biprocesses from above to ProVerif, it turns out that another simplification is required before the tool terminates: we need to add a tag to encryptions, preventing an encryption to be used as input to evaluation more than once. More specifically, if an encryption is created using $\mathsf{encrypt}_{T,ek,crs}$ then it contains a **countone** tag; and when an encryption goes through eval_e the tag must be **countone** and is changed to **countzero**. This restriction is sound for this particular protocol and enough for ProVerif to terminate.

Under this simplified model we have successfully analysed the protocol and found that in all three cases the equivalences are satisfied, and hence the protocol realises the OT functionality.

As sanity checks we also tried variations of the protocols to see if ProVerif would find the expected flaws that would then arise. If **two** also becomes a member of type $T = bit$ then ProVerif finds an attack in all three corruption scenarios. If the private decryption key dk_R is leaked then ProVerif finds an attack when both players are honest or when only R is honest. If the check that $\mathbf{encOf}_1(c_x) = c'_b$ in the receiver is omitted then ProVerif finds an attack when only R is honest.

[49] Algorithmically the let definition of a variable could be unrolled and the reduction rules be used to simply the conditions until they are trivial.

$$
\begin{aligned}
B_{OT}^{SR} \doteq\ & \mathrm{in}[in_{OT}^{R}, b];\\
& \quad \text{if } \mathbf{inType}_{bit}(b) = \mathbf{true} \text{ then}\\
& \quad \text{new } r_b;\\
& \quad \text{let } c_b' = \mathbf{enc}(\mathrm{choice}[b \cdot \mathbf{zero}], r_b, ek_R);\\
& \quad \text{let } p_b = \mathbf{proof}_{bit}(c_b', ek_R, crs_R);\\
& \quad \text{let } c_b = \mathbf{encPack}(c_b', ek_R, p_b, crs_R);\\
& \quad \mathrm{out}[leak_{RS}, c_b];\\
& \mathrm{in}[in_{OT}^{S}, x_{01}];\\
& \quad \text{if } \mathbf{isPair}(x_{01}) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_0 = \mathbf{first}(x_{01});\\
& \quad \text{let } x_1 = \mathbf{second}(x_{01});\\
& \quad \text{if } \mathbf{isValue}(x_0) = \mathbf{true} \text{ then}\\
& \quad \text{if } \mathbf{isValue}(x_1) = \mathbf{true} \text{ then}\\
& \quad \text{new } r, r_0, r_1;\\
& \quad \text{let } c_x' = \mathbf{eval}_{sel}(c_b', \mathrm{choice}[x_0 \cdot \mathbf{zero}], \mathrm{choice}[x_1 \cdot \mathbf{zero}], r);\\
& \quad \text{let } d_0 = \mathbf{com}(\mathrm{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S);\\
& \quad \text{let } d_1 = \mathbf{com}(\mathrm{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S);\\
& \quad \text{let } p_x = \mathbf{proof}_{sel}(c_x', \ldots, crs_S);\\
& \quad \text{let } c_x = \mathbf{evalPack}(c_x', \ldots, crs_S);\\
& \quad \mathrm{out}[leak_{SR}, c_x];\\
& \mathrm{in}[infl_{SR}, \mathbf{deliver}];\\
& \quad \text{let } x_b = \mathrm{choice}[\mathbf{dec}(c_x', dk_R) \cdot \mathbf{zero}];\\
& \quad \text{if } \mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true} \text{ then}\\
& \quad\quad \mathrm{out}\big[out_{OT}^{R}, \mathrm{choice}[x_b \cdot x_0]\big]\\
& \quad \text{else}\\
& \quad\quad \mathrm{out}\big[out_{OT}^{R}, \mathrm{choice}[x_b \cdot x_0]\big]
\end{aligned}
$$

Figure 4.101: Biprocess B_{OT}^{SR} for when both are honest

$Q^S_{left} \doteq \text{in}[receive_{RS}, c_b];$
if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
if $\mathbf{ekOf}(c_b) = ek_R$ then
if $\mathbf{crsOf}(c_b) = crs_R$ then
$\text{out}[out^S_{OT}, \mathbf{getInput}];$
$\text{in}[in^S_{OT}, x_{01}];$
if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
let $x_0 = \mathbf{first}(x_{01});$
let $x_1 = \mathbf{second}(x_{01});$
if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
new $r, r_0, r_1;$
let $c'_b = \mathbf{encOf}(c_b);$
let $c'_x = \mathbf{eval}_{sel}(c'_b, x_0, x_1, r);$
let $d_0 = \mathbf{com}(x_0, r_0, ck_S);$
let $d_1 = \mathbf{com}(x_1, r_1, ck_S);$
let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S);$
let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S);$
$\text{out}[send_{SR}, c_x]$

$Q^S_{right} \doteq \text{in}[receive_{RS}, c_b]$
if $\mathbf{verEncPack}_{bit}(c_b) = \mathbf{true}$ then
if $\mathbf{ekOf}(c_b) = ek_R$ then
if $\mathbf{crsOf}(c_b) = crs_R$ then
let $b = \mathbf{extractEnc}(\mathbf{proofOf}(c_b), extd_R);$
if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
$\text{out}[out^S_{OT}, \mathbf{getInput}];$
$\text{in}[in^S_{OT}, x_{01}];$
if $\mathbf{isPair}(x_{01}) = \mathbf{true}$ then
let $x_0 = \mathbf{first}(x_{01});$
let $x_1 = \mathbf{second}(x_{01});$
if $\mathbf{isValue}(x_0) = \mathbf{true}$ then
if $\mathbf{isValue}(x_1) = \mathbf{true}$ then
if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
new $r, r_0, r_1;$
let $c'_b = \mathbf{encOf}(c_b);$
let $c'_x = \mathbf{enc}(x_0, r, ek_R);$
let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S);$
let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S);$
let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S);$
let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S);$
$\text{out}[send_{SR}, c_x]$
else
new $r, r_0, r_1;$
let $c'_b = \mathbf{encOf}(c_b);$
let $c'_x = \mathbf{enc}(x_1, r, ek_R);$
let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S);$
let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S);$
let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S);$
let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S);$
$\text{out}[send_{SR}, c_x]$

Figure 4.102: The merged and simplified processes from the real (left) and ideal (right) protocol when only S is honest

$$
\begin{array}{l}
B^S_{OT} \doteq \text{in}[\mathit{receive}_{RS}, c_b] \\
\quad \text{if } \mathbf{verEncPack}_{bit}(c_b) = \mathbf{true} \text{ then} \\
\quad \text{if } \mathbf{ekOf}(c_b) = ek_R \text{ then} \\
\quad \text{if } \mathbf{crsOf}(c_b) = crs_R \text{ then} \\
\quad \text{let } b = \mathbf{extractEnc}(\mathbf{proofOf}(c_b), extd_R); \\
\quad \text{out}[out^S_{OT}, \mathbf{getInput}]; \\
\quad \text{in}[in^S_{OT}, x_{01}]; \\
\quad \text{if } \mathbf{isPair}(x_{01}) = \mathbf{true} \text{ then} \\
\quad \text{let } x_0 = \mathbf{first}(x_{01}); \\
\quad \text{let } x_1 = \mathbf{second}(x_{01}); \\
\quad \text{if } \mathbf{isValue}(x_0) = \mathbf{true} \text{ then} \\
\quad \text{if } \mathbf{isValue}(x_1) = \mathbf{true} \text{ then} \\
\quad \text{if } \mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true} \text{ then} \\
\quad\quad \text{new } r, r_0, r_1; \\
\quad\quad \text{let } c'_b = \mathbf{encOf}(c_b); \\
\quad\quad \text{let } c'_x = \text{choice}\big[\mathbf{eval}_{sel}(c'_b, x_0, x_1, r) \cdot \mathbf{enc}(x_0, r, ek_R)\big]; \\
\quad\quad \text{let } d_0 = \mathbf{com}(\text{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S); \\
\quad\quad \text{let } d_1 = \mathbf{com}(\text{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S); \\
\quad\quad \text{let } p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S); \\
\quad\quad \text{let } c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S); \\
\quad\quad \text{out}[\mathit{send}_{SR}, c_x] \\
\quad\quad \text{else} \\
\quad\quad\quad \text{new } r, r_0, r_1; \\
\quad\quad\quad \text{let } c'_b = \mathbf{encOf}(c_b); \\
\quad\quad\quad \text{let } c'_x = \text{choice}\big[\mathbf{eval}_{sel}(c'_b, x_0, x_1, r) \cdot \mathbf{enc}(x_1, r, ek_R)\big]; \\
\quad\quad\quad \text{let } d_0 = \mathbf{com}(\text{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S); \\
\quad\quad\quad \text{let } d_1 = \mathbf{com}(\text{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S); \\
\quad\quad\quad \text{let } p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S); \\
\quad\quad\quad \text{let } c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S); \\
\quad\quad\quad \text{out}[\mathit{send}_{SR}, c_x]
\end{array}
$$

Figure 4.103: Biprocess B^S_{OT} for when only S is honest

$Q^R_{left} \doteq \text{in}[in^R_{OT}, b];$
if $\textbf{inType}_{bit}(b) = \textbf{true}$ then
new r_b;
let $c'_b = \textbf{enc}(b, r_b, ek_R)$;
let $p_b = \textbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
let $c_b = \textbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\text{out}[send_{RS}, c_b]$;
$\text{in}[receive_{SR}, c_x]$;
if $\textbf{verEvalPack}_{sel}(c_x) = \textbf{true}$ then
if $\textbf{encOf}_1(c_x) = c'_b$ then
if $\textbf{ekOf}(c_x) = ek_R$ then
if $\textbf{ckOf}(c_x) = ck_S$ then
if $\textbf{crsOf}(c_b) = crs_S$ then
let $x_b = \textbf{dec}(\textbf{encOf}(c_x), dk_R)$;
$\text{out}[out^R_{OT}, x_b]$

$Q^R_{right} \doteq \text{in}[in^R_{OT}, b];$
if $\textbf{inType}_{bit}(b) = \textbf{true}$ then
new r_b;
let $c'_b = \textbf{enc}(\textbf{zero}, r_b, ek_R)$;
let $p_b = \textbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
let $c_b = \textbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\text{out}[send_{RS}, c_b]$;
$\text{in}[receive_{SR}, c_x]$;
if $\textbf{verEvalPack}_{sel}(c_x) = \textbf{true}$ then
if $\textbf{encOf}_1(c_x) = c'_b$ then
if $\textbf{ekOf}(c_x) = ek_R$ then
if $\textbf{ckOf}(c_x) = ck_S$ then
if $\textbf{crsOf}(c_x) = crs_S$ then
let $x_0 = \textbf{extractEval}_1(\textbf{proofOf}(c_x), extd_S)$;
let $x_1 = \textbf{extractEval}_2(\textbf{proofOf}(c_x), extd_S)$;
if $\textbf{eqValue}(b, \textbf{zero}) = \textbf{true}$ then
$\text{out}[out^R_{OT}, x_0]$
else
$\text{out}[out^R_{OT}, x_1]$

Figure 4.104: The merged and simplified processes from the real (left) and ideal (right) protocol when only R is honest

$B^R_{OT} \doteq \text{in}[in^R_{OT}, b];$
if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
new r_b;
let $c'_b = \mathbf{enc}(\text{choice}[b \cdot \mathbf{zero}], r_b, ek_R)$;
let $p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R)$;
let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\text{out}[send_{RS}, c_b]$
$\text{in}[receive_{SR}, c_x]$;
if $\mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true}$ then
if $\mathbf{encOf}_1(c_x) = c'_b$ then
if $\mathbf{ekOf}(c_x) = ek_R$ then
if $\mathbf{ckOf}(c_x) = ck_S$ then
if $\mathbf{crsOf}(c_x) = crs_S$ then
let $x_b = \text{choice}\big[\mathbf{dec}(\mathbf{encOf}(c_x), dk_R) \cdot \mathbf{zero}\big]$;
let $x_0 = \text{choice}\big[\mathbf{zero} \cdot \mathbf{extractEval}_1(\mathbf{proofOf}(c_x), extd_S)\big]$;
let $x_1 = \text{choice}\big[\mathbf{zero} \cdot \mathbf{extractEval}_2(\mathbf{proofOf}(c_x), extd_S)\big]$;
if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
$\text{out}\big[out^R_{OT}, \text{choice}[x_b \cdot x_0]\big]$
else
$\text{out}\big[out^R_{OT}, \text{choice}[x_b \cdot x_1]\big]$

Figure 4.105: Biprocess B^R_{OT} when only R is honest

4.8 Remarks

We end with a few straight-forward extensions together with suggestions for future work addressing some of the shortcomings of this chapter.

4.8.1 Extentions

For presetational purposes we have left out the following easy extensions in the previous sections.

Hybrid analysis approach. As mentioned in the introduction it is also possible to use our result to analyse a broader class of protocols using a hybrid-symbolic approach. Here we are given a protocol Π on no particular form and may now analyse it in our framework as follows:

1. decompose it into a protocol π on the support form (ie. using only the supported primitives in the allowed ways) and a set of subprotocols $\Pi_1, \ldots, \Pi_n$ (on no particular form) that share no crypto with π nor with each other[50]
2. formulate ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_n$ on the supported form and show that the subprotocols $\Pi_1, \ldots, \Pi_n$ realise them
3. formulate target ideal functionality $\mathcal{G}$ and simulator Sim on the supported form
4. let Sys_{real} be the real protocol composed of π and $\mathcal{F}_1, \ldots, \mathcal{F}_n$, and let Sys_{ideal} be the ideal protocol composed of $\mathcal{G}$ and Sim; show in the symbolic model (possibly using ProVerif) that $\mathcal{S}(Sys_{real}) \overset{s}{\sim} \mathcal{S}(Sys_{ideal})$ holds
5. use the soundness theorem to conclude that $\mathcal{RW}(Sys_{real}) \overset{c}{\sim} \mathcal{RW}(Sys_{ideal})$, and in turn through composition, that Π realise $\mathcal{G}$ using the combined simulators

Note that in step 2. there are no requirements on whether or not Π_i can be shown to realise $\mathcal{F}_i$ in our framework: as long as $\mathcal{F}_i$ can be expressed in our model as an ideal functionality then this step may be done recursively through our framework, but it may also be done manually and using cryptographic primitives beyond those we support. Supporting ideal functionalities enables this kind of hybrid analysis.

As before we also still only need to consider one session of the protocol since the compositional theorem guarantees that it remains secure even when composed with itself a polynomial number of times.

Symbolic criteria. While the approach advocated in this work requires the manual construction of a simulator, our soundness results may also be used for the *symbolic criteria* approach where it is once and for all shown (possibly outside the framework) that if a protocol π satisfied a given symbolic condition then there exists a simulator the ensures indistinguishability relative to a fixed ideal functionality. This is for instance the approach taken in [CH06] where a symbolic criteria for a key agreement functionality is given and proved sound.

4.8.2 Future Work

The following suggestions would also be interesting to consider.

Probabilistic programmes. We have only considered deterministic programmes (in that the probabilistic choices are limited to the randomness used as input to the cryptographic primitives) yet many protocols, including protocols for multiparty computation and zero-knowledge

[50] Note that this is not a limitation of our framework as it also applies to eg. the UC framework where only information theoretical (and not computational) cryptography may be shared across ideal functionalities.

proofs, make fundamental use of probability as part of their security guarantees. By extending the protocol model to allow for programmes making probabilistic choices we may capture such protocols[51].

We have circumvented this problem in a few instances by allowing the random choices to be made by the environment: if a protocol is secure when all the random choices are done by the adversary then clearly it is also secure when these are instead drawn from a distribution. However this is a strictly stronger condition for some protocols, and the extra choices for the environment may slow down the automatic analysis significantly.

Moreover, it also means that we cannot use simpler expressions to describe the output values of ideal functionalities compared to the outputs of real protocols; in particular, the exact same output must be computed in both cases as everything is deterministic. Without this limitation we could for instance more clearly capture the essence of a multiplication protocol for additive shares by idealising (abstracting) even more. We furthermore cannot let an ideal functionality dictate that a value chosen by a realising protocol must be chosen at random.

To capture probabilistic programmes in a symbolic model we would need a probabilistic calculus and a probabilistic formulation of observational equivalence. It seems that the work of Goubault et al. [GPT07] might be a suitable choice allowing the soundness to carry over easily. One downside is that no automated tool exist for this calculus.

Note that another issue rises if the probabilisic choices are furthermore allowed to depend on the security parameter κ as it is not clear how to capture this in the symbolic model (currently κ does not exist in this model at all). Having this option would allow us to capture the full triple-generation protocol of [BDOZ11] where e is drawn from $\{0,1\}^\kappa$.

Variable-length programmes. Supporting programmes beyond the constant-length programmes used here would allow more protocols, including those for multi-party computations, to be analysed. One possible problem here is to ensure soundness of the symbolic interpretation, in particular in terms of polynomial running time as pointed out in [Unr11].

From two-party to multi-party. Although multi-party protocols may sometimes be naturally expressed as compositions of two-party protocols, it would still be interesting to add support for an arbritrary but fixed set of players (allowing a dynamic set of players seems likely to introduce even more problems).

If players are allowed to forward packages from other players then we must be careful that the translator can always extract. More specifically, we must for instance prevent that a corrupt player forms a package using the CRS of an honest player and sends this to another honest player; in this case the translator cannot extract as the CRS was generated for simulation, yet it is not clear how to reject such packages in a way that is also natural in the real-world interpretation of a real procotol.

Automatic process merging. In Section 4.7 we tried to be somewhat systematic in massaging the processes from the symbolic interpretation into biprocesses suitable for the ProVerif tool. A static analysis may be developed to properly automate this task of soundly simplifying and merging the processes from the symbolic interpretation into suitable biprocesses.

[51] At a technical level, one approach would be to allow programmes with several edges having the same ψ condition but annotated with a probability.

4.9 Bibliography

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 104–115, New York, NY, USA, 2001. ACM.

[AR02] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15:103–127, 2002.

[BAF05] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[Bla04] Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 86–100, May 2004.

[Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

[Bla11] Bruno Blanchet. *Cryptographic Protocol Verifier (ProVerif) User Manual, version 1.85*, 2011. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[BMM10] Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *FSTTCS*, volume 8 of *LIPIcs*, pages 352–363. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[BP03] Michael Backes and Birgit Pfitzmann. A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2003.

[BP04] Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable dolev-yao style cryptographic library. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 204 – 218, june 2004.

[BP06] Michael Backes and Birgit Pfitzmann. On the cryptographic key secrecy of the strengthened yahalom protocol. In *Security and Privacy in Dynamic Environments*, volume 201 of *IFIP International Federation for Information Processing*, pages 233–245. Springer US, 2006.

[BPW03] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 220–230, New York, NY, USA, 2003. ACM.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[Can05] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*, 2005. http://eprint.iacr.org/2000/067.

[Can08] Ran Canetti. Composable formal security analysis: Juggling soundness, simplicity and efficiency. In *ICALP*, volume 5126 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2008.

[CC08] Hubert Lundh Comon and Véronique Cortier. Computational soundness of observational equivalence. In *ACM Conference on Computer and Communications Security*, pages 109–118. ACM, 2008.

[CG10] Ran Canetti and Sebastian Gajek. Universally composable symbolic analysis of diffie-hellman based key exchange. *IACR Cryptology ePrint Archive*, 2010:303, 2010.

[CH06] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 380–403. Springer Berlin Heidelberg, 2006.

[CHKS12] Hubert Lundh Comon, Masami Hagiya, Yusuke Kawamoto, and Hideki Sakurada. Computational soundness of indistinguishability properties without computable parsing. In *Information Security Practice and Experience*, volume 7232 of *Lecture Notes in Computer Science*, pages 63–79. Springer Berlin Heidelberg, 2012.

[CKW11] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46:225–259, 2011.

[CW11] Veronique Cortier and Bogdan Warinschi. A composable computational soundness notion. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[DDMR07] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172(0):311–358, 2007.

[DKMR05] Anupam Datta, Ralf Küsters, John C. Mitchell, and Ajith Ramanathan. On the relationships between notions of simulation-based security. In *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 476–494. Springer Berlin Heidelberg, 2005.

[DKP09] Stéphanie Delaune, Steve Kremer, and Olivier Pereira. Simulation based security in the applied pi calculus. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 169–180, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DNO08] Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. Essentially optimal universally composable oblivious transfer. In *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2008.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer Berlin Heidelberg, 2012.

[GPT07] Jean Larrecq Goubault, Catuscia Palamidessi, and Angelo Troina. A probabilistic applied pi-calculus. In *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 175–190. Springer Berlin Heidelberg, 2007.

[LN08] Peeter Laud and Long Ngo. Threshold homomorphic encryption in the universally composable cryptographic library. In *Provable Security*, volume 5324 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin / Heidelberg, 2008.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1-3):118–164, 2006.

[MW04] Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer Berlin Heidelberg, 2004.

[PW01] Birgit. Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 184 –200, 2001.

[Unr11] Dominique Unruh. Termination-insensitive computational indistinguishability (and applications to computational soundness). In *CSF*, pages 251–265. IEEE Computer Society, 2011.