Universally Composable Symbolic Analysis for Two-Party Protocols based on Homomorphic Encryption

Morten Dahl Ivan Damgård

Aarhus University*

Abstract

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.

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.

*The authors acknowledge support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, and also from the CFEM research centre (supported by the Danish Strategic Research Council) within which part of this work was performed.

[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.

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).

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 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

[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.

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, $\mathcal{S}(Sys_1) \overset{s}{\sim} \mathcal{S}(Sys_2)$ 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

[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.

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 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 7 carry out an analysis of the oblivious transfer protocol from [DNO08].

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 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 5.4 and 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

[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.

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.

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].

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.

[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 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.

1.3 Organisation

The rest of the chapter is organised as shown next. As a reading hint, Section 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 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 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 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 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 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 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 8 we give a few remarks on possible extensions and future work.

1.4 Acknowledgements

We would like to thank Ran Canetti for valuable discussion and insights, and for hosting Morten at BU in the beginning of this work. We would also like to thank Hubert Comon-Lundh for discussion and clarification of his work in [CC08]. Finally, we are thankful for the valuable feedback provided by anonymous reviewers, including mentioning of several places that needed clarification.

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.

2.1 Motivating Example

We introduce our protocol class by way of an running example. Consider the OT protocol from [DNO08] shown in Figure 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.

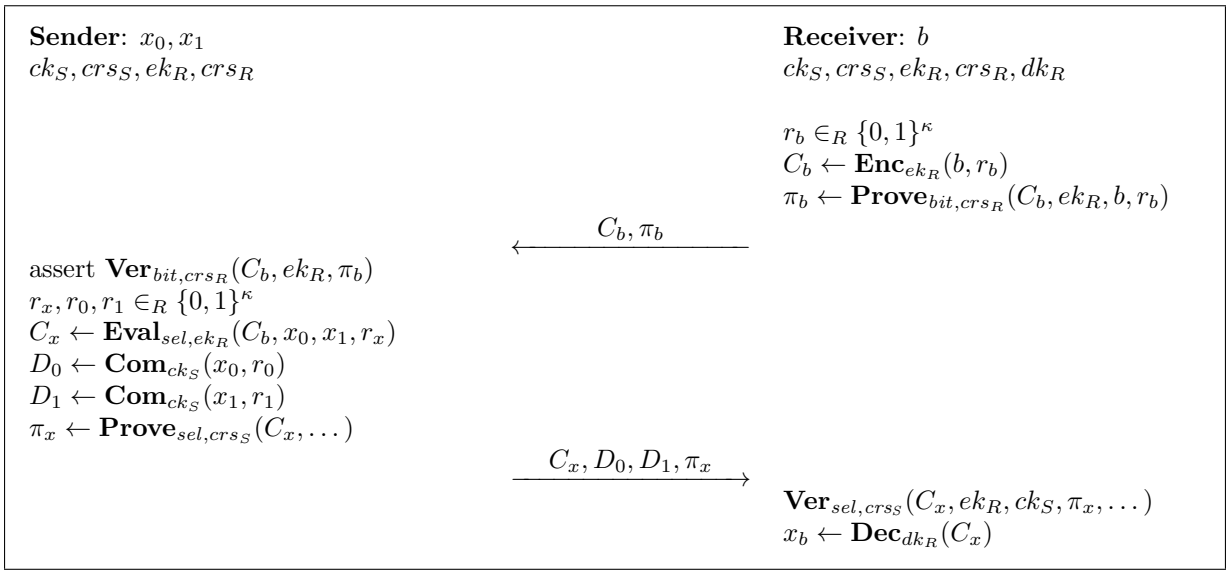

Figure 1: The OT protocol of [DNO08] in the original notation

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.

[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.

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 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.

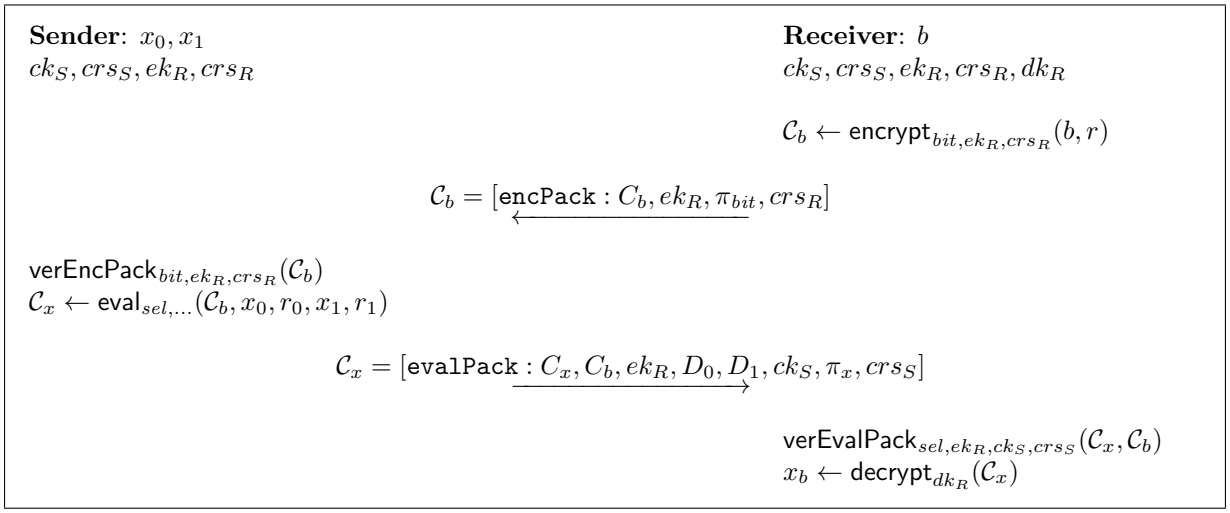

Figure 2: The OT protocol of [DNO08] in our annotated notation

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.

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

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).

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 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 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 $receive_{SR}$. Figure 4 shows the flow of the entire protocol.

Note that the sender programme P_{OT}^S 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_{OT}^S 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 $\text{input}_{\mathcal{P}}[p : x]$ 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 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 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 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[14]. Note also that the eval_e method (unlike commit_U

[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.

[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 $\mathsf{encrypt}_{bit,ek_R,crs_R}$.

[14] Although there is no explicit opening or decommitment operation allowing players to open a commitment in the typical way (by e.g. revealing both plaintext value and randomness used), this may not be a serious limitation: commitments can be opened by revealing the plaintext value and giving a NIZK proof that the value is correct, or an ideal commitment functionality that allows for opening can be used instead (and realised outside the framework). See Section 2.7.

$P^S_{OT} \doteq \text{input}_\emptyset[\mathit{receive}_{RS} : c_b];$
if $\mathsf{verEncPack}_{bit,R,R}(c_b)$ then
output$[\mathit{out}^S_{OT} : \texttt{getInput}]$;
$\text{input}_\emptyset[\mathit{in}^S_{OT} : x_{01}]$;
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$[\mathit{send}_{RS} : c_b]$;
$\text{input}_\emptyset[\mathit{receive}_{SR} : c_x]$;
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 3: Player programme P^S_{OT} for sender (left) and P^R_{OT} for receiver (right)

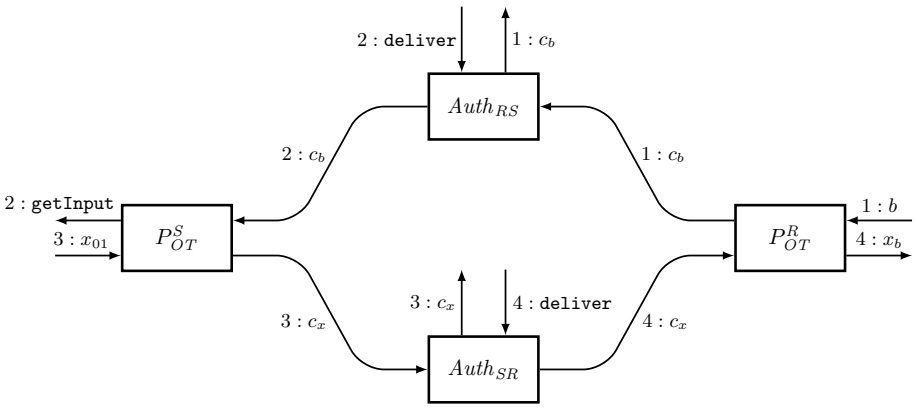

Figure 4: Linkes and message flow in the real OT protocol when both players are honest

$\mathsf{isValue}(x) \to b$ 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 5: Union of all operations available to the programmes in our protocol class

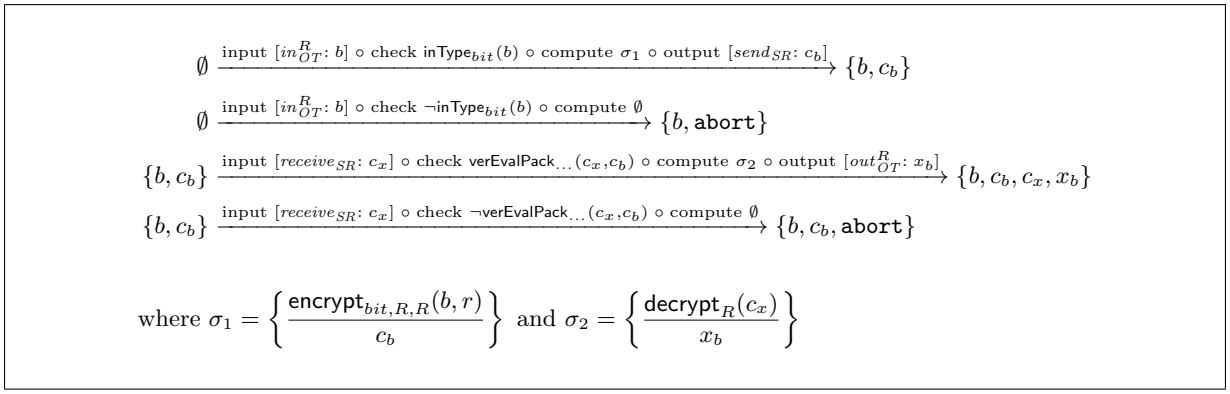

Figure 6: Formal player programme P^R_{OT} for OT receiver

and $\mathsf{encrypt}_T$) does not take a randomness 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 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[15], 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, $\upsilon_P : \mathbb{R}_P \to \mathbb{V} \times \{\mathtt{enc}, \mathtt{com}\}$, 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.

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[16]. 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.

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

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

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

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

Figure 7: Programme for an authenticated channel $Auth_{AB}$

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

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

Figure 8: Operations available to plain programmes

simple predefined channel programmes given in Figure 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 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 9. The keys and operations given to programme P^B for player B follows symmetrically.

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 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 three systems

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

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.

(as the operations for plain programmes in Figure 8...)

$$\begin{array}{rr} & \mathsf{isComPack}(x) \to b \\ \mathsf{decrypt}_{dk_A}(x) \to v & \mathsf{commit}_{U,ck_A,crs_A}(v, r) \to d \\ & \mathsf{verComPack}_{U,ck_B,crs_B}(d) \to b \end{array}$$

$$\begin{array}{rr} \mathsf{isEncPack}(x) \to b & \mathsf{isEvalPack}(x) \to b \\ \mathsf{encrypt}_{T,ek,crs_A}(v, r) \to c & \mathsf{eval}_{e,ek,ck_A,crs_A}(c_1, c_2, v_1, r_1, v_2, r_2) \to c \\ \mathsf{verEncPack}_{T,ek,crs_B}(c) \to b & \mathsf{verEvalPack}_{e,ek,ck_B,crs_B}(c, c_1, c_2) \to b \\ & \mathsf{verEvalPack}_{e,ek,ck_B,crs_B}(c, c_1, c_2, d_1, d_2) \to b \end{array}$$

Figure 9: Operations available to player programme P^A – with $ek \in \{ek_A, ek_B\}$

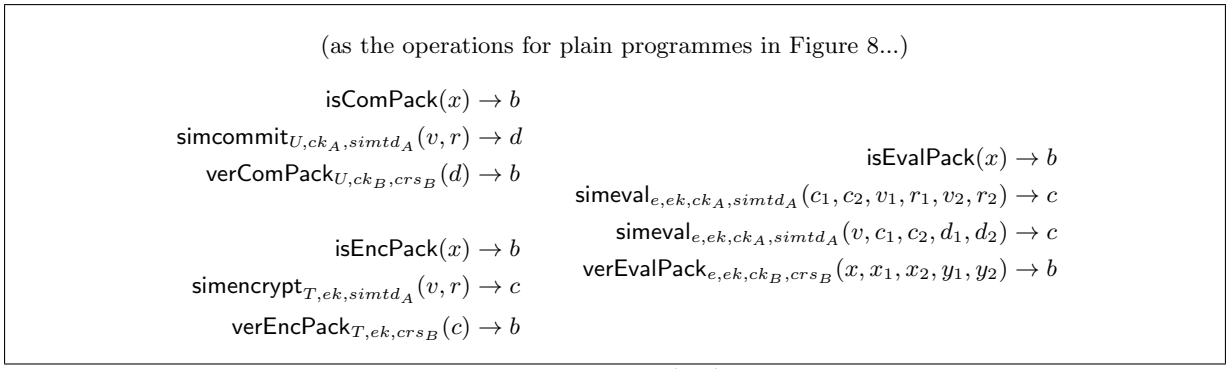

Figure 10: Operations available to simulator $Sim^{AB,A}$ when both honest – $ek \in \{ek_A, ek_B\}$

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 10, and symmetrically for the simulator for player B; in case only player A is honest the simulator Sim^{A} may use the operations in Figure 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[17].

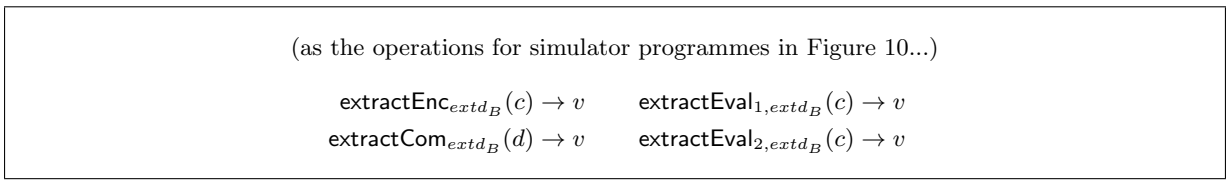

Figure 11: Operations available to simulator programme Sim^{A} when only A is honest

Definition 2.2 (Ideal protocol)**.** *Let the following components be given:*

- *a target functionality* $\mathcal{F} = \left(\mathcal{F}^{AB}, \mathcal{F}^{A}, \mathcal{F}^{B}\right)$
- *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 three systems

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

$$Sys^{A}_{ideal} \doteq \mathcal{F}^{A} \diamond Sim^{A} \diamond \mathcal{S}^{A} \qquad Sys^{B}_{ideal} \doteq \mathcal{F}^{B} \diamond Sim^{B} \diamond \mathcal{S}^{B}$$

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.*

[17] 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.

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 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 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 13. For the case where only S is honest we have the ideal functionality and simulator in Figure 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 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 7 we use ProVerif to conclude that the systems of these two triples are indistinguishable.

$\mathcal{F}^{SR}_{OT} \doteq$ 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

$Sim^{SR,S}_{OT} \doteq$ 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

$Sim^{SR,R}_{OT} \doteq$ 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 12: Ideal OT functionality and simulators for when both players are honest

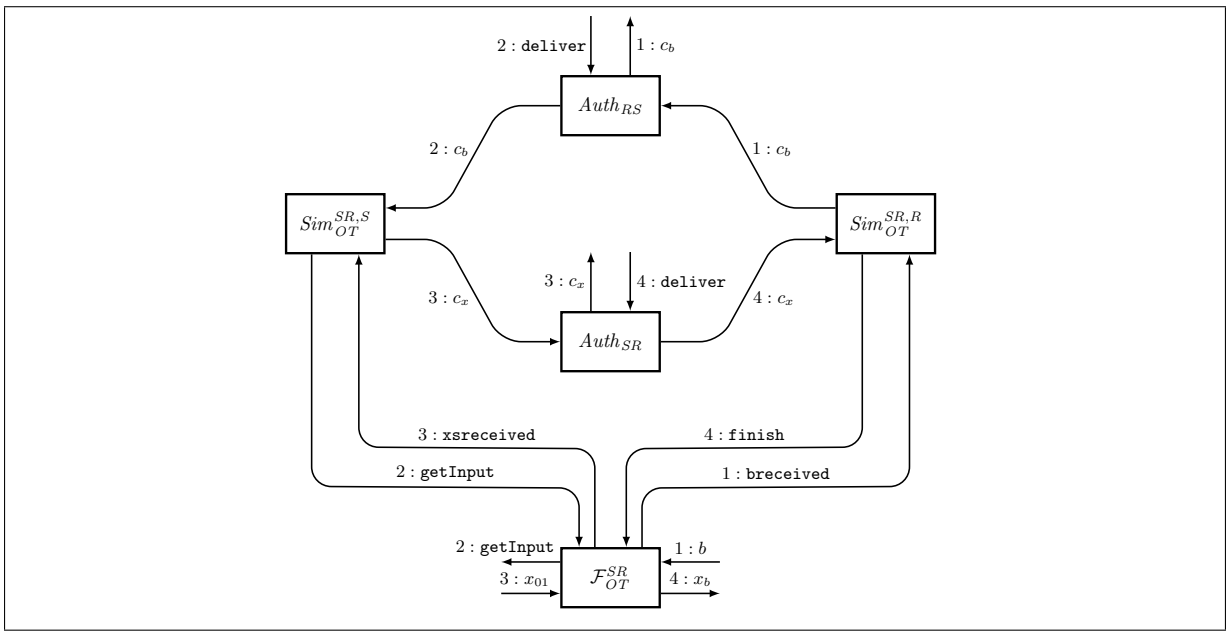

Figure 13: Links and message flow of ideal OT protocol when both players are honest

$\mathcal{F}_{OT}^{S} \doteq$ input$_\emptyset[infl_{OT} : b]$;
if $\mathsf{inType}_{bit}(b)$ then
output$[out_{OT}^{S} : \texttt{getInput}]$;
input$_\emptyset[in_{OT}^{S} : x_{01}]$;
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$[leak_{OT} : x_0]$;
stop
else
output$[leak_{OT} : x_1]$;
stop

$Sim_{OT}^{S} \doteq$ input$_\emptyset[receive_{RS} : c_b]$;
if $\mathsf{verEncPack}_{bit,R,R}(c_b)$ then
let $b \leftarrow \mathsf{extractEnc}_R(c_b)$;
output$[infl_{OT} : b]$;
input$_\emptyset[leak_{OT} : x_b]$;
let $c_x \leftarrow \mathsf{simeval}_{sel,R,S,S}(x_b, c_b, 0, r_0, 0, r_1)$;
output$[send_{SR} : c_x]$;
stop

Figure 14: Ideal OT functionality and simulator for when only S is honest

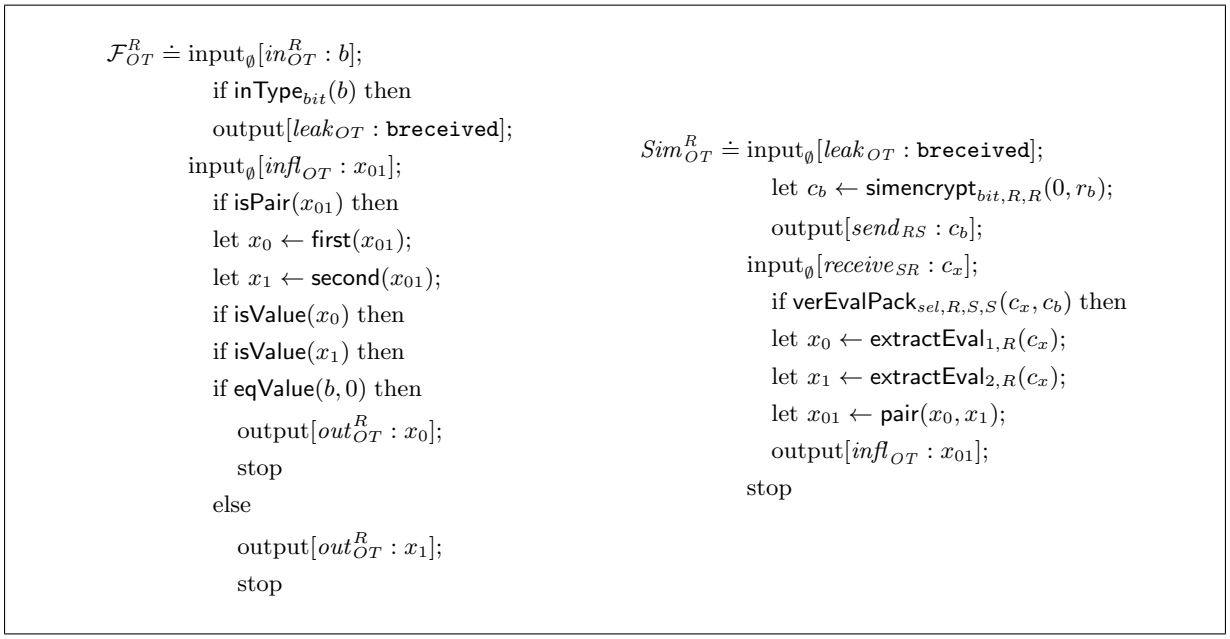

Figure 15: Ideal OT functionality and simulator for when only R is honest

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 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[18]. The real protocol becomes

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

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 17, 18, and Figure 19. The ideal protocol becomes

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

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

[18] 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.

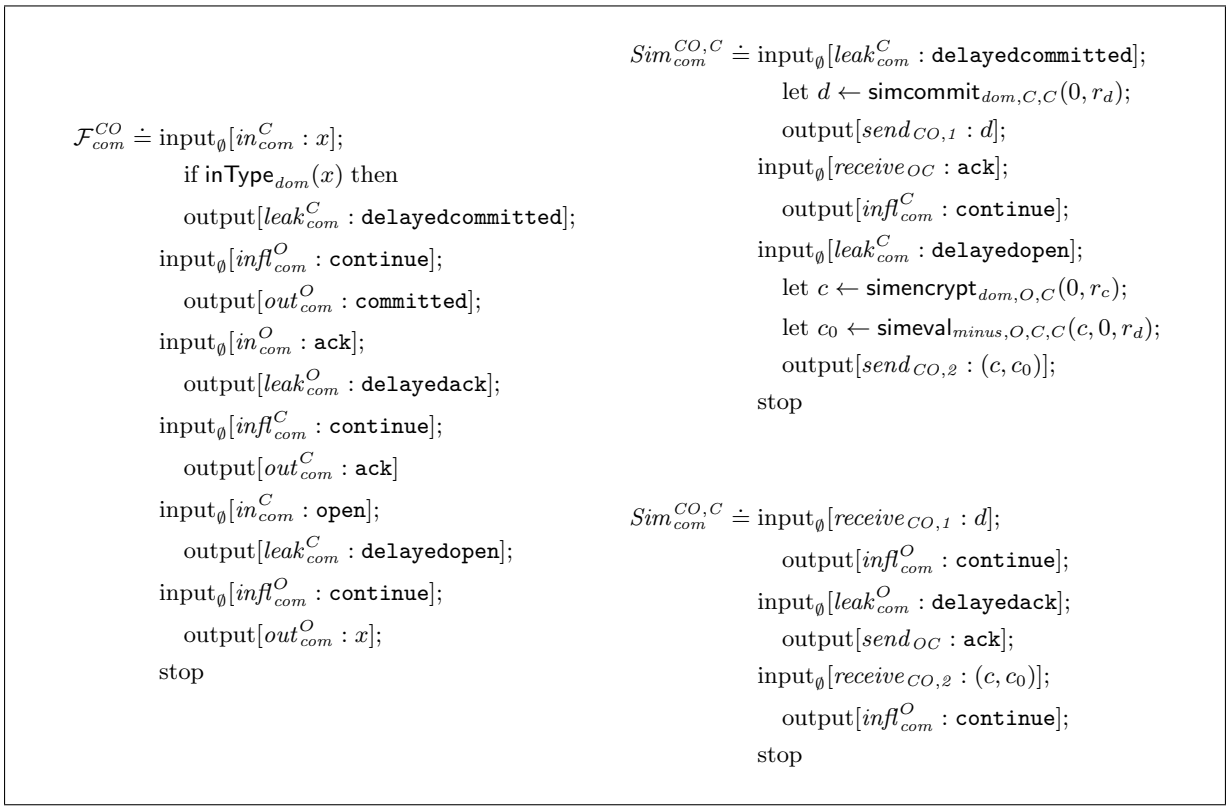

Figure 17: Ideal commitment functionality and simulators for when both players are honest

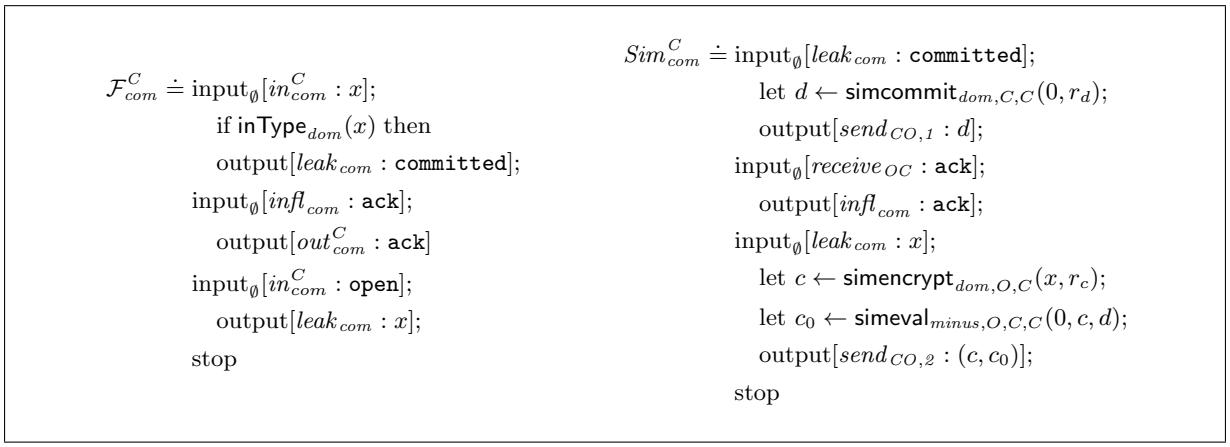

Figure 18: Ideal commitment functionality and simulator for when only committer is honest

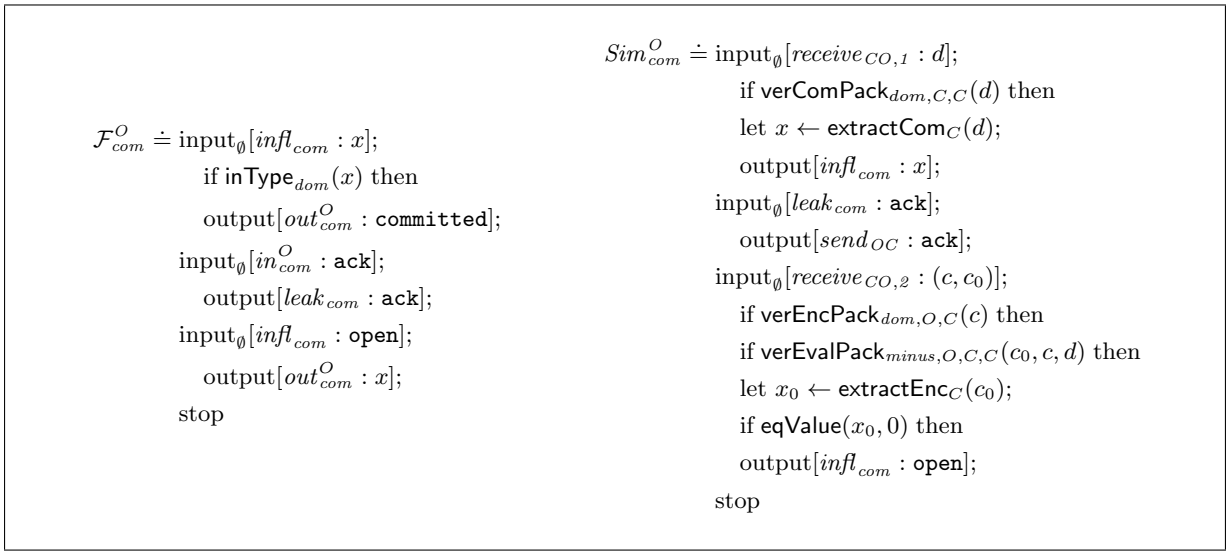

Figure 19: Ideal commitment functionality and simulator for when only opener is honest

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[19]. 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, and she computes 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 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 21, 22, and 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
- $\text{output}[in^C_{com} : d]$;
- $\text{input}_{\emptyset}[out^C_{com} : \texttt{ack}]$;
- $\text{output}[send_{AB} : \texttt{proceed}]$;
- $\text{input}_{\emptyset}[receive_{BA} : b]$;
- if $\mathsf{inType}_{bit}(b)$ then
- let $c \leftarrow \mathsf{peval}_{xor}(a, b)$;
- $\text{output}[out^A_{cf} : c]$;
- $\text{input}_{\emptyset}[in^A_{cf} : \texttt{outputToB}]$;
- $\text{output}[in^C_{com} : \texttt{open}]$;
- stop

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

Figure 20: Player programme P^A_{CF} for A (left) and P^B_{CF} for B (right)

[19] 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. However, if a protocol realises a functionality for inputs chosen by the environment, then it also does so when the inputs are instead drawn randomly from a distribution. Further discussion is given in Section 8.

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

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

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

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

Figure 21: Ideal CF functionality, simulators, and simulated functionality when both are honest

$$
\begin{aligned}
Sim^{A}_{CF} \doteq\ & \text{input}_{\emptyset}[leak_{cf} : \texttt{receivedInputFromA}];\\
& \quad \text{output}[in^{C}_{com} : \texttt{fakeCommit}];\\
& \text{input}_{\emptyset}[out^{C}_{com} : \texttt{ack}];\\
& \quad \text{output}[send_{AB} : \texttt{proceed}];\\
& \text{input}_{\emptyset}[receive_{BA} : b];\\
& \quad \text{if } \mathsf{inType}_{bit}(b) \text{ then}\\
& \quad \text{output}[infl_{cf} : b];\\
& \text{input}_{\emptyset}[leak_{cf} : a];\\
& \quad \text{output}[in^{C}_{com} : a];\\
& \text{stop}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}^{A}_{CF} \doteq\ & \text{input}_{\emptyset}[in^{A}_{cf} : a];\\
& \quad \text{if } \mathsf{inType}_{bit}(a) \text{ then}\\
& \quad \text{output}[leak_{cf} : \texttt{receivedInputFromA}];\\
& \text{input}_{\emptyset}[infl_{cf} : b];\\
& \quad \text{if } \mathsf{inType}_{bit}(b) \text{ then}\\
& \quad \text{let } c \leftarrow \mathsf{peval}_{xor}(a, b);\\
& \quad \text{output}[out^{A}_{cf} : c]\\
& \text{input}_{\emptyset}[in^{A}_{cf} : \texttt{outputToB}];\\
& \quad \text{output}[leak^{A}_{cf} : a];\\
& \text{stop}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}^{C}_{com} \doteq\ & \text{input}_{\emptyset}[in^{C}_{com} : \texttt{fakeCommit}];\\
& \quad \text{output}[leak_{com} : \texttt{committed}];\\
& \text{input}_{\emptyset}[infl_{com} : \texttt{ack}];\\
& \quad \text{output}[out^{C}_{com} : \texttt{ack}]\\
& \text{input}_{\emptyset}[in^{C}_{com} : x];\\
& \quad \text{output}[leak_{com} : x];\\
& \text{stop}
\end{aligned}
$$

Figure 22: Ideal CF functionality, simulator, and simulated functionality for only A honest

$$
\begin{aligned}
Sim^{B}_{CF} \doteq\ & \text{input}_{\emptyset}[out^{O}_{com} : (\texttt{committed}, a)];\\
& \quad \text{output}[in^{O}_{com} : \texttt{ack}];\\
& \text{input}_{\{out^{O}_{com}\}}[receive_{AB} : \texttt{proceed}];\\
& \quad \text{output}[infl_{cf} : a];\\
& \text{input}_{\{out^{O}_{com}\}}[leak_{cf} : b];\\
& \quad \text{output}[send_{BA} : b];\\
& \text{input}_{\emptyset}[out^{O}_{com} : \texttt{open}];\\
& \quad \text{output}[infl_{cf} : \texttt{outputToB}];\\
& \text{stop}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}^{B}_{CF} \doteq\ & \text{input}_{\emptyset}[infl_{cf} : a];\\
& \quad \text{if } \mathsf{inType}_{bit}(a) \text{ then}\\
& \quad \text{output}[out^{B}_{cf} : \texttt{getInput}];\\
& \text{input}_{\emptyset}[in^{B}_{cf} : b];\\
& \quad \text{if } \mathsf{inType}_{bit}(b) \text{ then}\\
& \quad \text{output}[leak_{cf} : b];\\
& \text{input}_{\emptyset}[infl_{cf} : \texttt{outputToB}];\\
& \quad \text{let } c \leftarrow \mathsf{peval}_{xor}(a, b);\\
& \quad \text{output}[out^{B}_{cf} : c];\\
& \text{stop}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}^{O}_{com} \doteq\ & \text{input}_{\emptyset}[infl_{com} : x];\\
& \quad \text{if } \mathsf{inType}_{bit}(x) \text{ then}\\
& \quad \text{output}[out^{O}_{com} : (\texttt{committed}, x)];\\
& \text{input}_{\emptyset}[in^{O}_{com} : \texttt{ack}];\\
& \quad \text{output}[leak_{com} : \texttt{ack}]\\
& \text{input}_{\emptyset}[infl_{com} : \texttt{open}];\\
& \quad \text{output}[out^{O}_{com} : \texttt{open}];\\
& \text{stop}
\end{aligned}
$$

Figure 23: Ideal CF functionality, simulator, and simulated functionality when only B honest

2.9 Multiplication Triple Functionality

As an exercise in expressibility[20] 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[21] 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 24 and 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[22] $\mathcal{C}_{a_1}$ 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 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 27 and 28; they simply execute the protocol using constants.

Figure 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 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 31 and its simulator in Figure 32. Here the error value e is extracted from player 1 instead.

[20] 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.

[21] While we may analyse some probabilistic choices, the kind used here falls into another category which it is unclear how to capture; see Section 8 for more discussion.

[22] 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 $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ to player 2
6. 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 $m(a_1) \leftarrow \mathsf{decrypt}_{dk_1}(\mathcal{C}_{m(a_1)})$
   (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 $\big(c_1, m(a_1), m(b_1), m(c_1)\big)$ on port out^1_{trip}

Figure 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 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 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 $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ to player 2
6. On receiving $\big(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\big)$ from player 2
   (a) influence `outputTo1` on port $infl^1_{trip}$

Figure 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 $\big(\mathcal{C}_{m(a_2)}, \mathcal{C}_{m(b_2)}, \mathcal{C}_{m(c_2)}\big)$ 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 $\big(\mathcal{C}_{m(a_1)}, \mathcal{C}_{m(b_1)}, \mathcal{C}_{m(c_1)}\big)$ to player 1

Figure 28: Simulator $\mathit{Sim}^{12,2}_{trip}$ 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 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 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 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 32: Simulator Sim^2_{trip} when only player 2 is honest

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 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, i.e. may be simulated by a single polynomial time ITM given the security parameter κ and initial input z.

3.1 Commitment Scheme

A *commitment scheme* is given by PPT algorithms $\mathbf{ComKeyGen}(1^\kappa) \to ck$ and $\mathbf{Com}_{ck}(V, R) \to D$ 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. 33 with more than negligible probability (in the security parameter)
- **computationally binding:** if no PPT adversary $\mathcal{A}$ may win the game in Figure 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 35

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

Note that here and below we require security for the entire domain of values so that the concrete length of a plaintext need not be leaked (a suitable length may be derived from the publicly known domain); this is justified by the fact that we consider homomorphic encryption where it seems natural that the cryptographic operations cannot be applied to pairs etc. but only to values.

1. Generate commitment key $ck \leftarrow \mathbf{ComKeyGen}(1^\kappa)$ 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 $D' \leftarrow \mathbf{Com}_{ck}(V, R)$
4. Adversary wins if $D = D'$

Figure 33: Security game $\mathcal{G}^{com,ws}$ for a *well-spread* commitment scheme

3.2 Homomorphic Encryption Scheme

An *encryption scheme* is given by three PPT algorithms $\mathbf{EncKeyGen}(1^\kappa) \to (ek, dk)$, $\mathbf{Enc}_{ek}(V, R) \to C$, and $\mathbf{Dec}_{dk}(C) \to V$. An *homomorphic* encryption scheme furthermore contains a PPT algorithm

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 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 b
4. Send D to the adversary

Figure 35: Security game $\mathcal{G}_b^{com,hi}$ for a *computationally hiding* commitment scheme

$\mathbf{Eval}_{e,ek}(C_1, C_2, V_1, V_2, R) \rightarrow C$ for arithmetic expression $e(x_1, x_2, y_1, y_2)$ and randomness R for re-randomisation[23]. 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 36 nor the game in Figure 37 with more than negligible probability
- **correct:** if no PPT adversary $\mathcal{A}$ may win neither the game in Figure 38 nor the game in Figure 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 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 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 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 36: Security game $\mathcal{G}^{enc,encws}$ for a *well-spread* encryption scheme

1. Generate encryption and decryption key $(ek, dk) \leftarrow \mathbf{EncKeyGen}(1^\kappa)$ and send both to 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 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 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 38: Security game $\mathcal{G}^{enc,enccor}$ for a *correct* encryption scheme

[23] 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 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 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 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 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 41: Security game $\mathcal{G}_b^{enc,cpa}$ for an IND-CPA secure encryption scheme

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 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 43
- **extractable:** if no PPT adversary $\mathcal{A}$ may win neither the game in Figure 44 nor the game in Figure 45 with more than negligible probability

and assume instantiations for:

- $\mathcal{R}_U = \big\{ (x, w) \,\big|\, D = \mathbf{Com}_{ck}(V, R) \wedge V \in U \big\}$ where $x = (D, ck)$ and $w = (V, R)$
- $\mathcal{R}_T = \big\{ (x, w) \,\big|\, C = \mathbf{Enc}_{ek}(V, R) \wedge V \in T \big\}$ where $x = (C, ek)$ and $w = (V, R)$
- $\mathcal{R}_e = \big\{ (x, w) \,\big|\, C = \mathbf{Eval}_{e,ek}(C_1, C_2, V_1, V_2, R) \wedge D_i = \mathbf{Com}_{ck}(V_i, R_i) \big\}$ 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 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. Repeat until adversary stops
   (a) Receive (x, w) from the adversary and check that $(x, w) \in \mathcal{R}$
   (b) 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 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 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 45: Security game $\mathcal{G}^{nizk,extract}$ for an *extractable* NIZK-PoK scheme

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.

4.1 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[24] $\mathcal{O}_P$ enforcing sanity checks on received messages and implementing the operations available to P as described in Section 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 46; the lines show the directional links between closed ports, and the dots represent open ports; Figure 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.

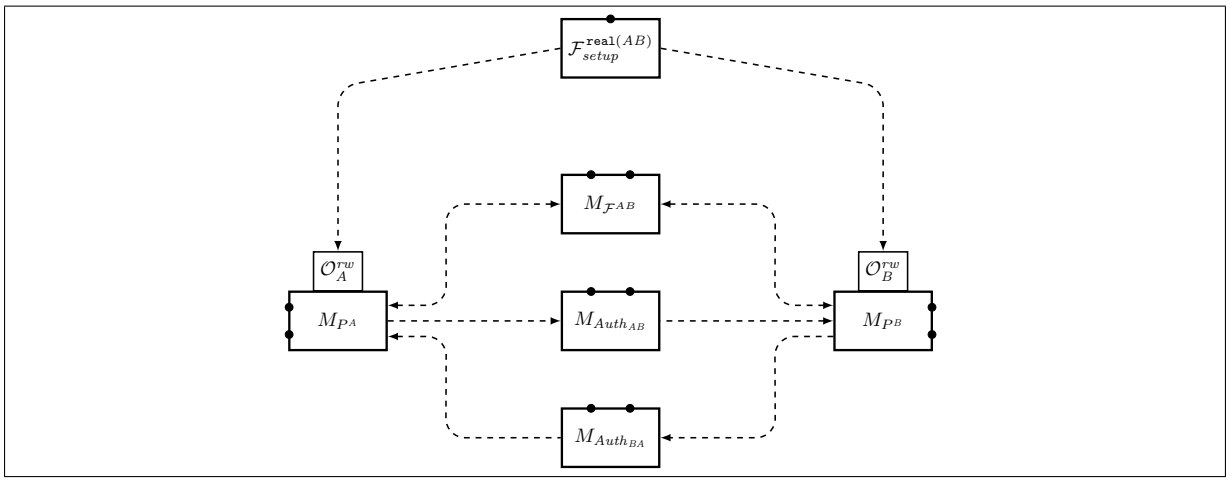

Figure 46: Real-world interpretation of example real protocol when both players are honest

Using ITMs defined in the remaining part of this section we may formalise the general structure as follows:

Definition 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.*

[24] 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.

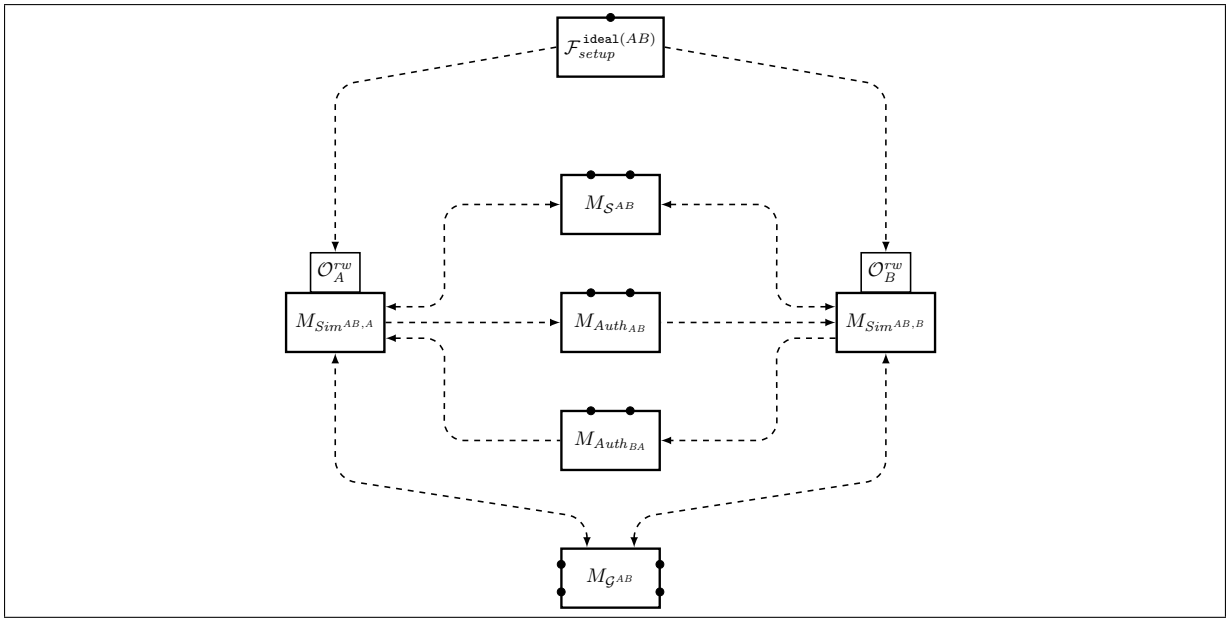

Figure 47: Real-world interpretation of example ideal protocol when both players are honest

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 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[25].

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

$$\psi = \mathsf{verEvalPack}_{sel, ek_B, ck_A, crs_A}(c, c_1, c_2, d_1, d_2)$$

[25] 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.

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.

4.3 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 48.

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 48: Real-world setup functionality $\mathcal{F}_{setup}$

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.

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 5.4: (i) to ensure that all encryptions in evaluation packages have the same encryption key

- $\texttt{storePlain}(BS) \to x$:
  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 49: Real-world implementation of well-formed checks for programmes

ek since the π_e proof does not guarantee this on its own[26]; (ii) to ensure that the C_1, C_2 encryptions of evaluation packages are already known to the player[27]; and (iii) to reject certain packages that an honest player would never have produced and which cannot occur in the intermediate interpretation[28].

The methods for storing received messages are then implemented in Figure 49. Methods for plain operations are implemented in Figure 50, and for commitments, encryptions and evaluations in Figure 51, 52 and 53. Finally, Figure 54 gives the implementations for decryption and extraction operations.

[26] 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 $\mathbf{Eval}_e$ procedure though as C_1 and C_2 are just bitstrings.

[27] 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).

[28] 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.

- $\texttt{isConst}(x) \to B$: 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 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 51: Real-world implementation of commitment operations for programmes

- $\texttt{isEncPack}(x) \to B$: if $\mu(x)$ match with $[\texttt{encPack} : \dots]$ 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 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 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 54: Real-world implementation of decryption and extraction operations for programmes

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 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.

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[29]. 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[30] $(\texttt{proof}_U : H_D, ck, crs)$, $(\texttt{proof}_T : H_C, ek, crs)$, $(\texttt{proof}_e : H_C, H_{C_1}, H_{C_2}, ek, H_{D_1}, H_{D_2}, ck, crs)$; and package 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 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 $\mathcal{F}_{mem}$.

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 55, 56, 57, 58 and 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 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.

[29] 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.

[30] 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 $\texttt{false}$ then return $\texttt{abort}$
  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 $\texttt{false}$ then return $\texttt{abort}$
  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 $\texttt{true}$
  - H match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return $\texttt{true}$
  - H match $\langle \texttt{pair} : H_1, H_2 \rangle$: verify $\texttt{acceptPlain}(H_1)$ and $\texttt{acceptPlain}(H_2)$; return $\texttt{true}$
  - return $\texttt{false}$ 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 $\texttt{true}$
  - H match $\langle \texttt{const} : Cn \rangle$: verify that Cn may be parsed as a constant; return $\texttt{true}$
  - 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 $\texttt{true}$
  - $\gamma(H)$ match $(\texttt{comPack} : H_D, ck, H_\pi, crs)$: verify that $\gamma(H_\pi)$ match $(\texttt{proof}_U : H_D, ck, crs)$; return $\texttt{true}$
  - $\gamma(H)$ 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 $\texttt{true}$
  - $\gamma(H)$ 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 $\texttt{true}$
  - return $\texttt{false}$ 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 55: Intermediate implementation of storing etc. for programmes

- $\texttt{isConst}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{const} : \dots \rangle$ then return $\texttt{true}$ else $\texttt{false}$
- $\texttt{eqConst}_{Cn}(x) \to B$: if $\mu(x)$ match with $\langle \texttt{const} : Cn \rangle$ return $\texttt{true}$ else $\texttt{false}$
- $\texttt{isValue}(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : \dots \rangle$ return $\texttt{true}$ else $\texttt{false}$
- $\texttt{eqValue}(v_1, v_2) \to B$: if $\mu(v_i)$ match $\langle \texttt{value} : V_i \rangle$ for $i \in [2]$, and $V_1 = V_2$, return $\texttt{true}$ else $\texttt{false}$
- $\texttt{inType}_U(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : V \rangle$, and V is in type U, return $\texttt{true}$ else $\texttt{false}$
- $\texttt{inType}_T(v) \to B$: if $\mu(v)$ match with $\langle \texttt{value} : V \rangle$, and V is in type T, return $\texttt{true}$ else $\texttt{false}$
- $\texttt{peval}_f(v_1, v_2, w_1, w_2) \to v$: 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 $\texttt{true}$ else $\texttt{false}$
- $\texttt{pair}(x_1, x_2) \to x$: 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 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 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 58: Intermediate implementation of encryption packages

- $\texttt{isEvalPack}(x) \to B$: if $\gamma(\mu(x))$ match with $(\texttt{evalPack} : \dots)$ return $\texttt{true}$ else $\texttt{false}$
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2) \to B$:
  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 $\texttt{true}$ else $\texttt{false}$
- $\texttt{verEvalPack}_{e,ek,ck,crs}(c, c_1, c_2, d_1, d_2) \to B$:
  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 $\texttt{true}$ else $\texttt{false}$
- $\texttt{eval}_{e,ek,ck,simtd}(c_1, c_2, w_1, r_1, w_2, r_2) \to c$:
  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 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 60: Intermediate implementation of decryption and extraction operations

$$\begin{aligned}
\texttt{isConst}(H) &\to B & \texttt{pair}(H_1, H_2) &\to H \\
\texttt{isValue}(H) &\to B & \texttt{first}(H) &\to H_1 \\
\texttt{isPair}(H) &\to B & \texttt{second}(H) &\to H_2 \\
\texttt{isComPack}(H) &\to B & \texttt{verComPack}_U(H) &\to B \\
\texttt{isEncPack}(H) &\to B & \texttt{verEncPack}_T(H) &\to B \\
\texttt{isEvalPack}(H) &\to B & \texttt{verEvalPack}_e(H) &\to B
\end{aligned}$$

$$\begin{aligned}
\texttt{commit}_{U,ck,crs}(V, R) &\to H & \texttt{eval}_{e,ck,ek,crs}(H_{C_1}, H_{C_2}, V_1, R_1, V_2, R_2) &\to H \\
\texttt{encrypt}_{T,ek,crs}(V, R) &\to H & \texttt{decrypt}_{dk}(H_C) &\to V
\end{aligned}$$

$\texttt{comOf}(H) \to H_D$ 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 61: Methods offered by the adversary's operation functionality $\mathcal{O}_{adv}$

5.3 The Adversary's Operation Module

All methods shown in Figure 61 are offered to the adversary by his operation module[31], 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.

[31] The operations given to the adversary are determined by the methods needed by the translator $\mathcal{T}^\star$ in Section 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.

- 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 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 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 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 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 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 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 62: Translator $\mathcal{T}^{AB}_{true,real}$

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[32]. 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.[33].

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 $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}^{\mathcal{H}}_{\star}[\mathcal{I}(Sys^{\mathcal{H}})]$ 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 $\mathcal{T}^{\mathcal{H}}_{\star}[\mathcal{I}(Sys^{\mathcal{H}})]$ is indistinguishable from $\mathcal{T}^{\mathcal{H}}_{true,real}[\mathcal{I}(Sys^{\mathcal{H}})]$, and the result follows.

Let $\mathcal{T}^{AB}_{true,real}$ be the translator defined in Figure 62. This translator emulates a setup functionality $\mathcal{F}_{setup}$ running in $\texttt{ideal}$ 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

[32] 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.

[33] 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.

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 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 64: Translator $\mathcal{T}^{AB}_{true,real}$ – encryption package from honest player

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 5.2. *We have* $\mathcal{RW}(Sys^{AB}) \overset{c}{\sim} \mathcal{T}^{AB}_{true,real}[\mathcal{I}(Sys^{AB})]$.

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 $\mathcal{T}^{AB}_{true,real}$ must translate handle H presenting a commitment package (Figure 63). Assume first that $\tau(H) = \perp$ 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, $\tau(H) = \perp$ 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 $\tau(H) \neq \perp$ we need to argue that resending the same translation

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 65: Translator $\mathcal{T}^{AB}_{true,real}$ – evaluation package from honest player

- 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 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 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 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 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 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 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 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 67: Translator $\mathcal{T}^A_{true,real}$ – commitment package from honest player

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 64) or an evaluation package (Figure 65) are similar. □

Next, let $\mathcal{T}^A_{true,real}$ be the translator defined in Figure 66. This translator also emulates the global memory functionality and the setup functionality $\mathcal{F}_{setup}$ running in $\texttt{ideal}$ 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 5.3. *We have* $\mathcal{RW}(Sys^A) \overset{c}{\sim} \mathcal{T}^A_{true,real}\big[\mathcal{I}(Sys^A)\big]$.

Proof. The argument goes in much the same way as for when both players are honest, relying on the logic of $\texttt{storeCrypto}$ 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

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 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 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 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 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 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 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 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 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 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 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 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 73: Translator $\mathcal{T}_\star^{AB}$

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 identity just needs to be preserved within each separation[34]: an encryption package is rejected (Figure 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 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. □

Let translator $\mathcal{T}_\star^{AB}$ be given by Figure 73 and translator $\mathcal{T}_\star^{A}$ by Figure 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

[34] 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 $D = \perp$ 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 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 $C = \perp$ 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 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 $D_i = \perp$ then
     (a) pick randomness $R_i \in \{0,1\}^\kappa$, compute $D_i \leftarrow \mathbf{Com}_{ck}(0, R)$, and update $\delta(H_{D_i}) \mapsto D_1$
  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 76: Translator $\mathcal{T}_\star^{AB}$ – evaluation package from honest player to honest 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 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 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 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 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 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 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 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 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 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 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 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 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 $H_{D_i} = \mathtt{comOf}_i(H)$ 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 $\delta(H_{D_i}) \mapsto D_1$
  2. for $i \in [2]$ let $C_i = \epsilon(\mathtt{encOf}_i(H))$
  3. pick fresh randomness $R \in \{0,1\}^\kappa$ and let $H_C = \mathtt{encOf}(H)$
  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 $\sigma(C, ek) \mapsto \mathtt{evalme}(H_C)$
  6. let $BS = [\mathtt{evalPackage} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$, store $\tau(H) = BS$, and return BS

Figure 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 $H_{D_i} = \mathtt{comOf}_i(H)$ 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 $\delta(H_{D_i}) \mapsto D_1$
  2. for $i \in [2]$ let $C_i = \epsilon(\mathtt{encOf}_i(H))$
  3. pick fresh randomness $R \in \{0,1\}^\kappa$, and let $H_C = \mathtt{encOf}(H)$ and $V = \mathtt{decrypt}_{dk}(H_C)$
  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 $\sigma(C, ek) \mapsto \mathtt{evalme}(H_C)$
  6. let $BS = [\mathtt{evalPackage} : C, C_1, ek, D_1, D_2, ck, \pi_e, crs]$, store $\tau(H) \mapsto BS$, return BS

Figure 82: Translator $\mathcal{T}_\star^A$ – evaluation package from honest player under corrupt key

Processing of bitstring $BS = [\mathtt{comPack} : D, ck, \pi_U, crs]$:

- 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 $H = \mathtt{commit}_{U,ck,crs}\big(V, D\big)$, and return H
- otherwise use $\mathtt{garbage}(\cdot)$ to create and return a garbage handle

Figure 83: Translator $\mathcal{T}_\star^A$ – commitment package from corrupt player

Processing of bitstring $BS = [\mathtt{encPack} : C, ek, \pi_T, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{T,crs}(C, ek, \pi_T)$ succeeds, and $\sigma(C, ek) \in \{\bot, \mathtt{encother}(\cdot)\}$ then:
  1. compute $(V, \cdot) \leftarrow \mathbf{Extract}_{T,extd}(C, ek, \pi_T)$, and let $H = \mathtt{encrypt}_{T,ek,crs}\big(V, C\big)$
  2. let $H_C = \mathtt{encOf}(H)$, store $\epsilon(H_C) \mapsto C$ and $\sigma(C, ek) \mapsto \mathtt{encother}(H_C)$, and return H
- otherwise use $\mathtt{garbage}(\cdot)$ to create and return a garbage handle

Figure 84: Translator $\mathcal{T}_\star^A$ – encryption package from corrupt player

Processing of bitstring $BS = [\mathtt{evalPack} : C, C_1, C_2, ek, D_1, D_2, ck, \pi_e, crs]$:

- if $ek \in \{ek_A, ek_B\}$, $\mathbf{Ver}_{e,crs}(C, \ldots, \pi_e)$ succ., and $\sigma(C, ek) = \mathtt{evalother}(BS, H, \cdot)$ 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 $\sigma(C_i, ek) \in \{\mathtt{encother}(H_{C_i}), \mathtt{evalother}(\cdot, \cdot, H_{C_i}), \mathtt{encme}(H_{C_i}), \mathtt{evalme}(H_{C_i})\}$ 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 $H = \mathtt{eval}_{e,ek,ck,crs}\big(H_{C_1}, H_{C_2}, W_1, D_1, W_2, D_2\big)$ and $H_C = \mathtt{encOf}(H)$
  3. store $\sigma(C, ek) \mapsto \mathtt{evalother}(BS, H, H_C)$ and $\epsilon(H_C) \mapsto C$, and return H
- otherwise use $\mathtt{garbage}(\cdot)$ to create and return a new garbage handle

Figure 85: Translator $\mathcal{T}_\star^A$ – evaluation package from corrupt player

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[35].

Lemma 5.4. *For any well-formed real protocol* $(Sys^{\mathcal{H}})_{\mathcal{H}}$ *we have* $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}^{\mathcal{H}}_{\star}\big[\mathcal{I}(Sys^{\mathcal{H}})\big]$.

Proof. Using the above results we proceed to show the result by a series of hybrid interpretations progressively changing $\mathcal{T}^{\mathcal{H}}_{true,real}$ into $\mathcal{T}^{\mathcal{H}}_{\star}$. Indistinguishability of the hybrids follows from the indistinguishability properties of the underlying cryptographic primitives. Note that through-out we use that well-formedness ensures that the same randomness is never used twice; this is important for indistinguishability of the primitives.

- 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 5.4.

Lemma 5.5. *For any well-formed ideal protocol* $(Sys^{\mathcal{H}})_{\mathcal{H}}$ *we have* $\mathcal{RW}(Sys^{\mathcal{H}}) \overset{c}{\sim} \mathcal{T}^{\mathcal{H}}_{\star}\big[\mathcal{I}(Sys^{\mathcal{H}})\big]$.

We can then state the soundness result.

Theorem 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* $\mathcal{RW}(Sys^{\mathcal{H}}_1) \overset{c}{\sim} \mathcal{RW}(Sys^{\mathcal{H}}_2)$.

[35] 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.

Proof. By assumption no polynomially bounded ITM $\mathcal{Z}'$ can tell the difference between $\mathcal{I}(Sys_1^{\mathcal{H}})$ and $\mathcal{I}(Sys_2^{\mathcal{H}})$ while having access only to $\mathcal{O}_{adv}$, in particular no $\mathcal{Z}' = \mathcal{Z} \diamond \mathcal{T}_\star^{\mathcal{H}}$ for a polynomially bounded ITM $\mathcal{Z}$. The result then follows by Lemma 5.4 and 5.5. □

Corollary 5.7. *Let $(Sys_{real}^{\mathcal{H}})_{\mathcal{H}}$ be a real protocol ϕ and let $(Sys_{ideal}^{\mathcal{H}})_{\mathcal{H}}$ be an ideal protocol with target functionality $\mathcal{F}$. If $\mathcal{I}(Sys_{real}^{\mathcal{H}}) \overset{c}{\sim} \mathcal{I}(Sys_{ideal}^{\mathcal{H}})$ 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 5.6. □

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[36], and indistinguishability. Moreover, we will later use that automated verification tools exist for this calculus in the form of ProVerif [Bla04, BAF05].

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[37] 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 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 86: Term constructor symbols

The destructor symbols are given in Figure 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[38]; 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 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 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[39] 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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

$\mathbf{isComPack}, \mathbf{verComPack}_U$	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 87: Term destructor symbols

nil	$\text{in}[p, x]; Q$	$\text{let } x = t \text{ in } Q \text{ else } Q'$	$Q \mid\mid Q'$
$\text{new } u; Q$	$\text{out}[p, t]; Q$	$\text{if } t = t' \text{ then } Q \text{ else } Q'$	$!\,Q$

Figure 88: Process syntax

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 89 and we only need a trivial equational theory[40]. 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 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 6.5 as symbolic indistinguishability

[40]The ProVerif manual [Bla11] advocates the use of rewrite rules over equations for efficiency reasons.

$$\mathbf{isValue}(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in \mathbf{Dom} \qquad \mathbf{isPair}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow \mathbf{true}$$
$$\mathbf{eqValue}(v, v) \rightsquigarrow \mathbf{true} \text{ for all } v \in \mathbf{Dom} \qquad \mathbf{first}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow x_1$$
$$\mathbf{inType}_U(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in U \qquad \mathbf{second}(\mathbf{pair}(x_1, x_2)) \rightsquigarrow x_2$$
$$\mathbf{inType}_T(v) \rightsquigarrow \mathbf{true} \text{ for all } v \in T \qquad \mathbf{eqConst}_c(c) \rightsquigarrow \mathbf{true} \text{ for all } c \in \mathbf{Const}$$
$$\mathbf{isConst}(c) \rightsquigarrow \mathbf{true} \text{ for all } c \in \mathbf{Const}$$

$$\mathbf{peval}_f(v_1, v_2, v_3, v_4) \rightsquigarrow v \text{ for all } v_i \in \mathbf{Dom} \text{ and } v = f(v_1, v_2, v_3, v_4)$$

$$\mathbf{isComPack}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verComPack}_U(\mathbf{comPack}(x_d, x_{ck}, \mathbf{proof}_U(x_d, x_{ck}, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$

$$\mathbf{isEncPack}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verEncPack}_T(\mathbf{encPack}(x_c, x_{ek}, \mathbf{proof}_T(x_c, x_{ek}, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$

$$\mathbf{isEvalPack}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{verEvalPack}_e(\mathbf{evalPack}(x_c, x_{c_1}, \ldots, x_{ck}, \mathbf{proof}_e(x_c, x_{c_1}, \ldots, x_{crs}), x_{crs})) \rightsquigarrow \mathbf{true}$$
$$\mathbf{eval}_e(\mathbf{enc}(v_1, x_{r_1}, \mathbf{ek}(x_{dk})), \mathbf{enc}(v_2, x_{r_2}, \mathbf{ek}(x_{dk})), v_3, v_4, x_r) \rightsquigarrow \mathbf{enc}(v, x_r, \mathbf{ek}(dk))$$
$$\text{for all } v_i \in \mathbf{Dom} \text{ and } v = \mathbf{peval}_e(v_1, v_2, v_3, v_4)$$
$$\mathbf{dec}(\mathbf{enc}(v, x_r, \mathbf{ek}(x_{dk})), x_{dk}) \rightsquigarrow v$$
$$\mathbf{extractCom}(\mathbf{proof}_U(\mathbf{com}(v, x_r, x_{ck}), x_{ck}, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v$$
$$\mathbf{extractEnc}(\mathbf{proof}_T(\mathbf{enc}(v, x_r, x_{ek}), x_{ek}, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v$$
$$\mathbf{extractEval}_i(\mathbf{proof}_e(\ldots, \mathbf{com}(v_1, x_{r_1}, x_{ck}), \mathbf{com}(v_2, x_{r_2}, x_{ck}), \ldots, \mathbf{crs}(x_{extd})), x_{extd}) \rightsquigarrow v_i \text{ for } i \in \{1, 2\}$$

$$\mathbf{comOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_d$$
$$\mathbf{ckOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ck}$$
$$\mathbf{proofOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{comPack}(x_d, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{encOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_c$$
$$\mathbf{ekOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_{ek}$$
$$\mathbf{proofOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{encPack}(x_c, x_{ek}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{encOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_c$$
$$\mathbf{encOf}_i(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{c_i} \text{ for } i \in \{1, 2\}$$
$$\mathbf{ekOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ek}$$
$$\mathbf{comOf}_i(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{d_i} \text{ for } i \in \{1, 2\}$$
$$\mathbf{ckOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{ck}$$
$$\mathbf{proofOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_\pi$$
$$\mathbf{crsOf}(\mathbf{evalPack}(x_c, x_{c_1}, x_{c_2}, x_{ek}, x_{d_1}, x_{d_2}, x_{ck}, x_\pi, x_{crs})) \rightsquigarrow x_{crs}$$

$$\mathbf{equals}(x, x) \rightsquigarrow \mathbf{true}$$

Figure 89: Term rewrite rules

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.

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[41]. 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 $\mathrm{out}[p^{call}_{\mathsf{encrypt}\ldots}, \mathbf{pair}(v,r)]; \mathrm{in}[p^{ret}_{\mathsf{encrypt}\ldots}, y]; Q$.

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

$$\mathrm{in}[p_1, x_1]; Q_{p_1} + \mathrm{in}[p_2, x_2]; Q_{p_2} + \cdots + \mathrm{in}[p_n, x_n]; Q_{p_n}$$

instead of

$$\text{new } p; \begin{pmatrix} \mathrm{out}[p, \mathbf{token}]; \mathrm{nil} \\ \quad || \ \mathrm{in}[p, x]; \mathrm{in}[p_1, x_1]; Q_{p_1} \\ \quad || \ \mathrm{in}[p, x]; \mathrm{in}[p_2, x_2]; Q_{p_2} \\ \quad || \ \ldots \\ \quad || \ \mathrm{in}[p, x]; \mathrm{in}[p_n, x_n]; Q_{p_n} \end{pmatrix}$$

for a process $\sum_i \mathrm{in}[p_i, x_i]; Q_{p_i}$ 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 $\mathrm{in}[p_i, x_i]; Q_{p_i}$ that may then finally be combined to obtain $Q = \sum_i \mathrm{in}[p_i, x_i]; Q_{p_i}$

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 90.

[41] 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.

$$
\begin{array}{l}
Q \doteq \text{in}[p_{in}, x_{in}]; \\
\quad \text{if } \mathsf{isEncPack}(x_{in}) = \textbf{true} \text{ then} \\
\quad\quad \text{if } \mathsf{verEncPack}_{T,ek,crs}(x_{in}) = \textbf{true} \text{ then} \\
\quad\quad\quad \text{let } y = \mathsf{decrypt}_{dk}(x_{in}); \\
\quad\quad\quad \text{out}[p_1, x_1]; Q_1 \\
\quad\quad \text{else } Q_3 \\
\quad \text{else if } \mathsf{isEvalPack}(x_{in}) = \textbf{true} \text{ then} \\
\quad\quad \text{if } \mathsf{verEvalPack}_{e,ek,ck,crs}(x_{in}) = \textbf{true} \text{ then} \\
\quad\quad\quad \text{let } y = \mathsf{decrypt}_{dk}(x_{in}); \\
\quad\quad\quad \text{out}[p_2, x_2]; Q_2 \\
\quad\quad \text{else } Q_3 \\
\quad \text{else } Q_3
\end{array}
$$

Figure 90: Example symbolic interpretation of programme point Σ

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 91, 92, and 93 respectively, where we have also omitted some "else" branches. Note that they follow the intermediate implementation in Section 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

$$Q_{box_P} \doteq !\, Q_{op_1} \mid\mid \cdots \mid\mid !\, Q_{op_n}$$

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

$$\text{new } p^{call}_{op_1}; \text{new } p^{ret}_{op_1}; \ldots; \text{new } p^{call}_{op_n}; \text{new } p^{ret}_{op_n}; \left(Q_P \mid\mid Q_{box_P}\right)$$

such that only Q_P may interact with Q_{box_P}.

As for the operations offered to the adversary (see Section 5.3) we first note that his operations may all be modelled as above. However, it is also sound to give the symbolic 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[42]

$$Q^{adv}_{box} \doteq !\, Q^{adv}_{\mathsf{commit}_U} \mid\mid !\, Q^{adv}_{\mathsf{encrypt}_T} \mid\mid !\, Q^{adv}_{\mathsf{eval}_e}$$

using the processes in Figure 95.

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 6.2 (Symbolic Interpretation)**.** *The* symbolic interpretation $\mathcal{S}(Sys)$ *of a well-formed system Sys is given by process* $\mathcal{E}^{\mathcal{H}}_{setup}\left[Q^{\mathcal{H}}_{honest} \mid\mid Q^{\mathcal{H}}_{adv}\right]$ *where the setup contexts* $\mathcal{E}^{\mathcal{H}}_{setup}$ *are given in Figure 96.*

[42] Note that there is no need to give different boxes in the different corruption scenarios.

$$
\begin{aligned}
Q_{\mathsf{verComPack}_{U,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{verComPack}_{U,ck,crs}}, x_d];\\
& \text{if } \mathbf{verComPack}_U(x_d) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{ckOf}(x_d) = ck \text{ then}\\
& \text{if } \mathbf{crsOf}(x_d) = crs \text{ then}\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verComPack}_{U,ck,crs}}, \mathbf{true}]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{commit}_{U,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{commit}_{U,ck,crs}}, (x_v, x_r)];\\
& \text{if } \mathbf{inType}_U(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_d = \mathbf{com}(x_v, x_r, ck);\\
& \quad \text{let } x_\pi = \mathbf{proof}_U(x_d, ck, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{commit}_{U,ck,crs}}, \mathbf{comPack}(x_d, ck, x_\pi, crs)]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{simcommit}_{U,ck,simtd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{simcommit}_{U,ck,simtd}}, (x_v, x_r)];\\
& \text{if } \mathbf{isValue}(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_d = \mathbf{com}(x_v, x_r, ck);\\
& \quad \text{let } x_\pi = \mathbf{proof}_U(x_d, ck, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{simcommit}_{U,ck,crs}}, \mathbf{comPack}(x_d, ck, x_\pi, crs)]
\end{aligned}
$$

Figure 91: Symbolic implementation of operations for commitment packages

$$
\begin{aligned}
Q_{\mathsf{verEncPack}_{T,ek,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{verEncPack}_{T,ek,crs}}, x_c];\\
& \text{if } \mathbf{verEncPack}_T(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{ekOf}(x_c) = ek \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verEncPack}_{T,ek,crs}}, \mathbf{true}]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{encrypt}_{T,ek,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{encrypt}_{T,ek,crs}}, (x_v, x_r)];\\
& \text{if } \mathbf{inType}_T(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_c = \mathbf{enc}(x_v, x_r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_T(x_c, ek, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{encrypt}_{T,ek,crs}}, \mathbf{encPack}(x_c, ek, x_\pi, crs)]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{simencrypt}_{T,ek,simtd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{simencrypt}_{T,ek,crs}}, (x_v, x_r)];\\
& \text{if } \mathbf{isValue}(x_v) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_c = \mathbf{enc}(x_v, x_r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_T(x_c, ek, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{simencrypt}_{T,ek,crs}}, \mathbf{encPack}(x_c, ek, x_\pi, crs)]
\end{aligned}
$$

Figure 92: Symbolic implementation of operations for encryption packages

$$
\begin{aligned}
Q_{\mathsf{verEvalPack}_{e,ek,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{verEvalPack}_{e,ek,ck,crs}}, (x_c, x_{c_1}, x_{c_2}, x_{d_1}, x_{d_2})];\\
& \text{if } \mathbf{verEvalPack}_e(x_c) = \mathbf{true} \text{ then}\\
& \text{if } \mathbf{encOf}_i(x_c) = \mathbf{encOf}(c_i) \text{ then}\\
& \text{if } \mathbf{comOf}_i(x_c) = \mathbf{comOf}(d_i) \text{ then}\\
& \text{if } \mathbf{ekOf}(x_c) = ek \text{ then}\\
& \text{if } \mathbf{ckOf}(x_c) = ck \text{ then}\\
& \text{if } \mathbf{crsOf}(x_c) = crs \text{ then}\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{verEvalPack}_{e,ek,ck,crs}}, \mathbf{true}]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{eval}_{e,ek,ck,crs}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{eval}_{e,ek,ck,crs}}, (x_{c_1}, x_{c_2}, x_{v_1}, x_{r_1}, x_{v_2}, x_{r_2})];\\
& \text{if } \mathbf{ekOf}(x_{c_i}) = ek \text{ then}\\
& \text{if } \mathbf{isValue}(x_{v_i}) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_{c'_i} = \mathbf{encOf}(x_{c_i});\\
& \quad \text{let } x_{d'_i} = \mathbf{com}(x_{v_i}, x_{r_i}, ck);\\
& \quad \text{new } r;\\
& \quad \text{let } x_{c'} = \mathbf{eval}_e(x_{c'_1}, x_{c'_2}, x_{v_1}, x_{v_2}, r);\\
& \quad \text{let } x_\pi = \mathbf{proof}_e(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, crs);\\
& \quad \text{let } x_c = \mathbf{evalPack}(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, x_\pi, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{eval}_{e,ek,ck,crs}}, x_c]
\end{aligned}
$$

$$
\begin{aligned}
Q_{\mathsf{simeval}_{e,ek,ck,simtd}} \doteq\ & \mathrm{in}[p^{call}_{\mathsf{eval}_{e,ek,ck,crs}}, (x_v, x_{c_1}, x_{c_2}, x_{d_1}, x_{d_2})];\\
& \text{if } \mathbf{ekOf}(x_{c_i}) = ek \text{ then}\\
& \text{if } \mathbf{ckOf}(x_{d_i}) = ck \text{ then}\\
& \quad \text{let } x_{c'_i} = \mathbf{encOf}(x_{c_i});\\
& \quad \text{let } x_{d'_i} = \mathbf{comOf}(x_{d_i});\\
& \quad \text{new } r;\\
& \quad \text{let } x_{c'} = \mathbf{enc}(x_v, r, ek);\\
& \quad \text{let } x_\pi = \mathbf{proof}_e(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, crs);\\
& \quad \text{let } x_c = \mathbf{evalPack}(x_{c'}, x_{c'_1}, x_{c'_2}, ek, x_{d'_1}, x_{d'_2}, ck, x_\pi, crs);\\
& \quad \mathrm{out}[p^{ret}_{\mathsf{eval}_{e,ek,ck,crs}}, x_c]
\end{aligned}
$$

Figure 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 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 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 96: Setup evaluation contexts

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 turn essentially boils down to ensuring that the scheduling that leads to symbolic equivalence coincides with the scheduling policy used in the computational interpretations[43].

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[44]. 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 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[45] 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 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

[43] 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.

[44] 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.

[45] 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.

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 $M^0 = \langle \mathtt{pair} : H_1, H_2 \rangle$ 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[46] 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[47]. 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. □

[46] 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.

[47] 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).

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.

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 97, and the processes for the ideal functionality and simulators when both players honest are shown in Figure 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 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 **inType**$_{bit}(b) =$ **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 **isPair**$(x_{01}) =$ **true** then
let $x_0 =$ **first**(x_{01});
let $x_1 =$ **second**(x_{01});
if **isValue**$(x_0) =$ **true** then
if **isValue**$(x_1) =$ **true** then
out$[leak_{OT}^{S}, \mathbf{xsReceived}]$;
in$[infl_{OT}^{R}, \mathbf{finish}]$;
if **eqValue**$(b, \mathbf{zero}) =$ **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 **verEncPack**$_{bit}(c_b) =$ **true** then
if **ekOf**$(c_b) = ek_R$ then
if **crsOf**$(c_b) = crs_R$ then
out$[infl_{OT}^{S}, \mathbf{getInput}]$;
in$[leak_{OT}^{S}, \mathbf{xsReceived}]$;
let $c_b' =$ **encOf**(c_b);
let $c_x' =$ **eval**$_e(c_b', \mathbf{zero}, \mathbf{zero}, r)$;
let $d_0 =$ **com**$(\mathbf{zero}, r_0, ck_S)$;
let $d_1 =$ **com**$(\mathbf{zero}, r_1, ck_S)$;
let $p_x =$ **proof**$_{sel}(c_x', c_b', \ldots, crs_S)$;
let $c_x' =$ **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 $c_b' =$ **enc**$(\mathbf{zero}, r_b, ek_R)$;
let $p_b =$ **proof**$_{bit}(c_b', ek_R, crs_R)$;
let $c_b =$ **encPack**(c_b', ek_R, p_b, crs_R);
out$[send_{RS}, c_b]$;
in$[receive_{SR}, c_x]$;
if **verEvalPack**$_{sel}(c_x) =$ **true** then
if **encOf**$_1(c_x) = c_b'$ then
if **ekOf**$(c_x) = ek_R$ then
if **ckOf**$(c_x) = ck_S$ then
if **crsOf**$(c_x) = crs_S$ then
out$[infl_{OT}^{R}, \mathbf{finish}]$

Figure 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

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

$$\text{choice}[t_{left} \cdot t_{right}]$$

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[48] 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

$$\text{in}[\mathit{receive}_{RS}, c_b]; Q_0 \;||\; \text{in}[\mathit{send}_{SR}, x]; Q_1 \;||\; \text{in}[\mathit{send}_{RS}, x]; Q_2 \;||\; \text{in}[\mathit{in}^R_{OT}, b]; Q_3$$

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

$$\text{in}[\mathit{receive}_{RS}, c_b]; Q_0 \;||\; \text{in}[\mathit{send}_{SB}, x]; Q_1 \;||\; \text{in}[\mathit{infl}_{RS}, x]; Q'_2 \;||\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

representing the next state of protocol. This in turn leads to

$$\text{in}[\mathit{in}^S_{OT}, x_{01}]; Q'_0 \;||\; \text{in}[\mathit{send}_{SB}, x]; Q_1 \;||\; \text{nil} \;||\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

followed by

$$\text{nil} \;||\; \text{in}[\mathit{infl}_{SB}, x]; Q'_1 \;||\; \text{nil} \;||\; \text{in}[\mathit{receive}_{SR}, b]; Q'_3$$

and finally

$$\text{nil} \;||\; \text{nil} \;||\; \text{nil} \;||\; \text{nil}$$

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[49]; 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 99. For the ideal protocol we obtain the process in the left part of Figure 100.

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 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[50]. 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

[48] For readability we here present the procedure as working on processes instead of on programmes. An implementation could work on the programme trees instead.

[49] 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).

[50] Algorithmically the let definition of a variable could be unrolled and the reduction rules be used to simply the conditions until they are trivial.

$\text{in}[in^R_{OT}, b];$
$\quad$ 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 $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\quad$ $\text{out}[leak_{RS}, c_b]$;
$\quad$ $\text{in}[in^S_{OT}, x_{01}]$;
$\quad$ 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 $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
$\quad$ $\text{out}[leak_{SR}, c_x]$;
$\quad$ $\text{in}[infl_{SR}, \mathbf{deliver}]$;
$\quad$ let $x_b = \mathbf{dec}(c'_x, dk_R)$;
$\quad$ $\text{out}[out^R_{OT}, x_b]$

Figure 99: The merged processes from the real protocol, with the naive concatenation on the left and the simplified Q^{SR}_{left} on the right

$\text{in}[in^R_{OT}, b];$
$\quad$ if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
$\quad$ new r_b;
$\quad$ let $c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R)$;
$\quad$ 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 $c'_x = \mathbf{eval}_e(c'_b, \mathbf{zero}, \mathbf{zero}, r)$;
$\quad$ let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S)$;
$\quad$ let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S)$;
$\quad$ 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 $\mathbf{crsOf}(c_x) = crs_S$ then
$\quad$ if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
$\quad\quad$ $\text{out}[out^R_{OT}, x_0]$
$\quad$ else
$\quad\quad$ $\text{out}[out^R_{OT}, x_1]$

$Q^{SR}_{right} \doteq \text{in}[in^R_{OT}, b];$
$\quad$ if $\mathbf{inType}_{bit}(b) = \mathbf{true}$ then
$\quad$ new r_b;
$\quad$ let $c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R)$;
$\quad$ let $p_b = \mathbf{proof}_{bit}(c_b, ek_R, crs_R)$;
$\quad$ let $c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R)$;
$\quad$ $\text{out}[leak_{RS}, c_b]$;
$\quad$ $\text{in}[in^S_{OT}, x_{01}]$;
$\quad$ 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 $c'_x = \mathbf{eval}_e(c'_b, \mathbf{zero}, \mathbf{zero}, r)$;
$\quad$ let $d_0 = \mathbf{com}(\mathbf{zero}, r_0, ck_S)$;
$\quad$ let $d_1 = \mathbf{com}(\mathbf{zero}, r_1, ck_S)$;
$\quad$ let $p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S)$;
$\quad$ let $c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S)$;
$\quad$ $\text{out}[leak_{SR}, c_x]$;
$\quad$ $\text{in}[infl_{SR}, \mathbf{deliver}]$;
$\quad$ if $\mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true}$ then
$\quad\quad$ $\text{out}[out^R_{OT}, x_0]$
$\quad$ else
$\quad\quad$ $\text{out}[out^R_{OT}, x_1]$

Figure 100: The merged processes from the ideal protocol, with the naive concatenation on the left and the simplified Q^{SR}_{right} on the right

$$
\begin{aligned}
B^{SR}_{OT} \doteq\ & \text{in}[in^{R}_{OT}, b];\\
& \quad \text{if } \mathbf{inType}_{bit}(b) = \mathbf{true} \text{ then}\\
& \quad \text{new } r_b;\\
& \quad \text{let } c'_b = \mathbf{enc}(\text{choice}[b \cdot \mathbf{zero}], r_b, ek_R);\\
& \quad \text{let } p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R);\\
& \quad \text{let } c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R);\\
& \quad \text{out}[leak_{RS}, c_b];\\
& \text{in}[in^{S}_{OT}, x_{01}];\\
& \quad \text{if } \mathbf{isPair}(x_{01}) = \mathbf{true} \text{ then}\\
& \quad \text{let } x_0 = \mathbf{first}(x_{01});\\
& \quad \text{let } x_1 = \mathbf{second}(x_{01});\\
& \quad \text{if } \mathbf{isValue}(x_0) = \mathbf{true} \text{ then}\\
& \quad \text{if } \mathbf{isValue}(x_1) = \mathbf{true} \text{ then}\\
& \quad \text{new } r, r_0, r_1;\\
& \quad \text{let } c'_x = \mathbf{eval}_{sel}(c'_b, \text{choice}[x_0 \cdot \mathbf{zero}], \text{choice}[x_1 \cdot \mathbf{zero}], r);\\
& \quad \text{let } d_0 = \mathbf{com}(\text{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S);\\
& \quad \text{let } d_1 = \mathbf{com}(\text{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S);\\
& \quad \text{let } p_x = \mathbf{proof}_{sel}(c'_x, \ldots, crs_S);\\
& \quad \text{let } c_x = \mathbf{evalPack}(c'_x, \ldots, crs_S);\\
& \quad \text{out}[leak_{SR}, c_x];\\
& \text{in}[infl_{SR}, \mathbf{deliver}];\\
& \quad \text{let } x_b = \text{choice}[\mathbf{dec}(c'_x, dk_R) \cdot \mathbf{zero}];\\
& \quad \text{if } \mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true} \text{ then}\\
& \quad\quad \text{out}\big[out^{R}_{OT}, \text{choice}[x_b \cdot x_0]\big]\\
& \quad \text{else}\\
& \quad\quad \text{out}\big[out^{R}_{OT}, \text{choice}[x_b \cdot x_0]\big]
\end{aligned}
$$

Figure 101: Biprocess B^{SR}_{OT} for when both are honest

Figure 99, and a similar reasoning allows us to soundly simplify the ideal protocol to process Q^{SR}_{right} in the right part of Figure 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 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 102 that may be merged to form biprocess B^{S} in Figure 103. The same holds for when only R is honest; in this case Q^{R}_{left} and Q^{R}_{right} from Figure 104 yields biprocess B^{R} shown in Figure 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).

$Q^S_{left} \doteq \text{in}[receive_{RS}, c_b];$
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 102: The merged and simplified processes from the real (left) and ideal (right) protocol when only S is honest

$B^S_{OT} \doteq \text{in}[\mathit{receive}_{RS}, c_b]$
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 $b = \mathbf{extractEnc}(\mathbf{proofOf}(c_b), extd_R)$;
$\text{out}[out^S_{OT}, \mathbf{getInput}]$;
$\text{in}[in^S_{OT}, x_{01}]$;
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 $c'_x = \text{choice}\big[\mathbf{eval}_{sel}(c'_b, x_0, x_1, r) \cdot \mathbf{enc}(x_0, r, ek_R)\big]$;
let $d_0 = \mathbf{com}(\text{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S)$;
let $d_1 = \mathbf{com}(\text{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S)$;
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 $c'_x = \text{choice}\big[\mathbf{eval}_{sel}(c'_b, x_0, x_1, r) \cdot \mathbf{enc}(x_1, r, ek_R)\big]$;
let $d_0 = \mathbf{com}(\text{choice}[x_0 \cdot \mathbf{zero}], r_0, ck_S)$;
let $d_1 = \mathbf{com}(\text{choice}[x_1 \cdot \mathbf{zero}], r_1, ck_S)$;
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 103: Biprocess B^S_{OT} for when only S is honest

$$
\begin{array}{ll}
 & Q^R_{right} \doteq \text{in}[in^R_{OT}, b]; \\
 & \quad \text{if } \mathbf{inType}_{bit}(b) = \mathbf{true} \text{ then} \\
Q^R_{left} \doteq \text{in}[in^R_{OT}, b]; & \quad \text{new } r_b; \\
\quad \text{if } \mathbf{inType}_{bit}(b) = \mathbf{true} \text{ then} & \quad \text{let } c'_b = \mathbf{enc}(\mathbf{zero}, r_b, ek_R); \\
\quad \text{new } r_b; & \quad \text{let } p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R); \\
\quad \text{let } c'_b = \mathbf{enc}(b, r_b, ek_R); & \quad \text{let } c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R); \\
\quad \text{let } p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R); & \quad \text{out}[send_{RS}, c_b]; \\
\quad \text{let } c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R); & \text{in}[receive_{SR}, c_x]; \\
\quad \text{out}[send_{RS}, c_b]; & \quad \text{if } \mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true} \text{ then} \\
\text{in}[receive_{SR}, c_x]; & \quad \text{if } \mathbf{encOf}_1(c_x) = c'_b \text{ then} \\
\quad \text{if } \mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true} \text{ then} & \quad \text{if } \mathbf{ekOf}(c_x) = ek_R \text{ then} \\
\quad \text{if } \mathbf{encOf}_1(c_x) = c'_b \text{ then} & \quad \text{if } \mathbf{ckOf}(c_x) = ck_S \text{ then} \\
\quad \text{if } \mathbf{ekOf}(c_x) = ek_R \text{ then} & \quad \text{if } \mathbf{crsOf}(c_x) = crs_S \text{ then} \\
\quad \text{if } \mathbf{ckOf}(c_x) = ck_S \text{ then} & \quad \text{let } x_0 = \mathbf{extractEval}_1(\mathbf{proofOf}(c_x), extd_S); \\
\quad \text{if } \mathbf{crsOf}(c_b) = crs_S \text{ then} & \quad \text{let } x_1 = \mathbf{extractEval}_2(\mathbf{proofOf}(c_x), extd_S); \\
\quad \text{let } x_b = \mathbf{dec}(\mathbf{encOf}(c_x), dk_R); & \quad \text{if } \mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true} \text{ then} \\
\quad \text{out}[out^R_{OT}, x_b] & \quad\quad \text{out}[out^R_{OT}, x_0] \\
 & \quad \text{else} \\
 & \quad\quad \text{out}[out^R_{OT}, x_1]
\end{array}
$$

Figure 104: The merged and simplified processes from the real (left) and ideal (right) protocol when only R is honest

$$
\begin{array}{l}
B^R_{OT} \doteq \text{in}[in^R_{OT}, b]; \\
\quad \text{if } \mathbf{inType}_{bit}(b) = \mathbf{true} \text{ then} \\
\quad \text{new } r_b; \\
\quad \text{let } c'_b = \mathbf{enc}(\text{choice}[b \cdot \mathbf{zero}], r_b, ek_R); \\
\quad \text{let } p_b = \mathbf{proof}_{bit}(c'_b, ek_R, crs_R); \\
\quad \text{let } c_b = \mathbf{encPack}(c'_b, ek_R, p_b, crs_R); \\
\quad \text{out}[send_{RS}, c_b] \\
\text{in}[receive_{SR}, c_x]; \\
\quad \text{if } \mathbf{verEvalPack}_{sel}(c_x) = \mathbf{true} \text{ then} \\
\quad \text{if } \mathbf{encOf}_1(c_x) = c'_b \text{ then} \\
\quad \text{if } \mathbf{ekOf}(c_x) = ek_R \text{ then} \\
\quad \text{if } \mathbf{ckOf}(c_x) = ck_S \text{ then} \\
\quad \text{if } \mathbf{crsOf}(c_x) = crs_S \text{ then} \\
\quad \text{let } x_b = \text{choice}\big[\mathbf{dec}(\mathbf{encOf}(c_x), dk_R) \cdot \mathbf{zero}\big]; \\
\quad \text{let } x_0 = \text{choice}\big[\mathbf{zero} \cdot \mathbf{extractEval}_1(\mathbf{proofOf}(c_x), extd_S)\big]; \\
\quad \text{let } x_1 = \text{choice}\big[\mathbf{zero} \cdot \mathbf{extractEval}_2(\mathbf{proofOf}(c_x), extd_S)\big]; \\
\quad \text{if } \mathbf{eqValue}(b, \mathbf{zero}) = \mathbf{true} \text{ then} \\
\quad\quad \text{out}\big[out^R_{OT}, \text{choice}[x_b \cdot x_0]\big] \\
\quad \text{else} \\
\quad\quad \text{out}\big[out^R_{OT}, \text{choice}[x_b \cdot x_1]\big]
\end{array}
$$

Figure 105: Biprocess B^R_{OT} when only R is honest

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.

8 Remarks

We end with a few straight-forward extensions together with suggestions for future work addressing some of the shortcomings of this chapter.

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[51]
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.

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 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[52].

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.

[51] 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.

[52] At a technical level, one approach would be to allow programmes with several edges having the same ψ condition but annotated with a probability.

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) the distributions from which the shares are drawn. 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 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.

References

[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.