**EasyUC User Guide**

Riverside Research

Version 2.0

July 2024

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under N66001-22-C-4020 and HR001122C0185.  
  
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA and NIWC Pacific.

**Revision Sheet**

|  |  |  |
| --- | --- | --- |
| **Release No.** | **Date** | **Revision Description** |
| 1.0 | 12/16/2022 | Initial Release |
| 1.1 | 1/6/2023 | Removed EasyCrypt implementation sections (The Rest of the Story)  Added section on EPDPs |
| 1.2 | 3/17/2023 | Added distribution statement to footers  Clarified wording in several places  Updated to reflect changes to the UC DSL:   * Initial states cannot receive adversarial messages * Initial states in ideal functionalities must send adversarial messages * State machines cannot transition to the initial state * Parties are no longer required to serve a direct interface |
| 1.3 | 05/15/2023 | Clarified wording in several places.  Updated to reflect changes to the UC DSL:   * Destination port expressions in send statements must be in parentheses unless they are identifiers. |
| 2.0 | 07/15/2024 | Added a section on the UC DSL interpreter |
|  |  |  |

**EasyUC User Guide**

**TABLE OF CONTENTS**

[1 Introduction 1](#_Toc171931100)

[2 Preliminaries 1](#_Toc171931101)

[2.1 Obtaining EasyUC 1](#_Toc171931102)

[2.2 Running the UC DSL Type Checker 2](#_Toc171931103)

[2.3 Running the UC DSL Interpreter 3](#_Toc171931104)

[2.4 Running in Proof General 3](#_Toc171931105)

[3 Using EasyCrypt Theories 3](#_Toc171931106)

[4 Interfaces 4](#_Toc171931107)

[5 Ideal Functionalities 6](#_Toc171931108)

[6 Real Functionalities 10](#_Toc171931109)

[7 Simulators 14](#_Toc171931110)

[8 Composing Functionalities 18](#_Toc171931111)

[9 The UC DSL Interpreter 25](#_Toc171931112)

[9.1 Script Prelude 25](#_Toc171931113)

[9.2 Variables and Assumptions 27](#_Toc171931114)

[9.3 Real World Sessions 28](#_Toc171931115)

[9.3.1 Sending Messages (Port Indices) 29](#_Toc171931116)

[9.3.2 Executing the Functionality 31](#_Toc171931117)

[9.3.3 Asserting Effects 38](#_Toc171931118)

[9.4 Ideal World Sessions 39](#_Toc171931119)

[9.5 Resolving Blocks 44](#_Toc171931120)

[10 UC DSL Quick Reference 49](#_Toc171931121)

[10.1 Lexical Notes 49](#_Toc171931122)

[10.2 UC DSL Syntax 50](#_Toc171931123)

[10.3 EPDPs 54](#_Toc171931124)

[10.4 UC DSL Interpreter Syntax 56](#_Toc171931125)

[11 References 59](#_Toc171931126)

**LIST OF FIGURES**

Figure 1 Forw State Machine 10

Figure 2 State Machines of the KEReal Parties Pt1 (top) and Pt2 (bottom) 13

Figure 3 KEReal Message Flow 14

Figure 4 KEIdeal State Machine 16

Figure 5 KESim State Machine 18

Figure 6 KEIdeal and KESim Message Flow 18

Figure 7 SMCIdeal State Machine 20

Figure 8 State Machines of the SMCReal Parties Pt1 (top) and Pt2 (bottom) 22

Figure 9 SMCReal(KE) Message Flow 22

Figure 10 SMCSim State Machine 24

Figure 11 SMCIdeal and SMCSim Message Flow 25

Figure 12 Interpreter Script for KeyExchange Model 26

# Introduction

This guide describes ongoing work on EasyUC [2], a flavor of universally composable (UC) security [1] with machine-checked proofs built on top of the EasyCrypt proof assistant [4]. The reader is assumed to be familiar with the proof assistant EasyCrypt. The EasyCrypt Reference Manual [5] and EasyCrypt User Guide [6] are useful resources. The reader is also assumed to be familiar with universal composability. All example code is taken from [3].

EasyUC has two parts. The first is a formalization of UC in EasyCrypt, so that UC theorems can be stated and verified by machine. A series of prototypes has been explored [2][3] and the work is ongoing. The second part is a domain-specific language, UC DSL, that more directly and abstractly represents the concepts of EasyUC, allowing one to build and verify a UC model without resorting to EasyCrypt. UC DSL tooling is in the research and development phase, but its basic syntax is reasonably stable. Tools available so far are a type checker and an interpreter/debugger, which allows a user to interact with a model to observe its behavior. A translator to EasyCrypt is in work and a theorem prover is planned. This document describes the UC DSL and the interpreter.

EasyUC imposes a few restrictions on “standard” UC:

* Functionalities are created statically and have a fixed and known message-passing structure that permits hierarchical addressing, rather than allowing functionalities to be created and destroyed dynamically or decide at run time with whom they will communicate.
* The internal structures of real functionalities are not globally accessible and do not share state with other real functionalities.
* Arbitrary adversaries are modeled directly rather than restricting attention to the dummy adversary model.

The UC DSL is analogous to the module language of EasyCrypt. Both rely on the EasyCrypt ambient logic to define the types and operators that are used in variable declarations and expressions. State machine message handlers are composed of statements whose syntax is very similar to statements in EasyCrypt procedures. Instead of modules and procedures, however, the top-level constructs in UC DSL are interfaces and functionalities, and instead of procedures calling other procedures, functionalities exchange messages.

Following some preliminaries in Section 2, Sections 3 through 8 describe the UC DSL, Section 9 describes the UC DSL interpreter and Section 10 provides a quick reference to the DSL and the interpreter commands. Section 11 lists references.

# Preliminaries

## Obtaining EasyUC

This assumes you have EasyCrypt installed [4] and wish to add the EasyUC library, examples, UC DSL type checker and UC DSL interpreter.

1. EasyUC is a GitHub project (<https://github.com/easyuc/EasyUC>)
2. Create and change to a folder of your choosing and execute the command

$ git clone <https://github.com/easyuc/EasyUC>

1. Read and follow the installation instructions in uc-dsl/README.md. The instructions below are current as of 28 Jun 2024.
2. Install OCaml prerequisite packages, for example using the opam package manager

$ opam install dune batteries bisect\_ppx

1. Obtain the full path of the EasyCrypt ‘theories’ folder. One way is to run the command

$ easycrypt config

and look for the load-path entry:

load-path:  
 <system>@/home/rpg/.opam/4.11.1/lib/easycrypt/theories

1. Configure and build the UC DSL type checker and interpreter

$ cd EasyUC/uc-dsl

$ ./configure  
EasyCrypt theories directory pathname? <enter the load-path as above>

# If you’ve built before (e.g., you are updating the version),  
# run the cleanup command:

$ ./build-cleanup

$ ./build

$ ./install-opam

1. (Optional but highly recommended) Follow the instructions for integrating EasyUC with the Emacs text editor and Proof General

## Running the UC DSL Type Checker

UC DSL code is stored in text files with the ".uc" file type/suffix. File names must start with an uppercase letter (they must be valid EasyCrypt theory names). The type checker is invoked from the command prompt with the ucdsl command. The syntax of the command is

ucdsl [options] file

For example,

$ cd EasyUC/uc-dsl/examples/smc-case-study

$ ucdsl SMC.uc

Additional usage information:

1. With no arguments, ucdsl prints help info.
2. Command-line options
   1. {-include, -I} <dir> Add directory <dir> to the search path for theories
   2. -raw-msg Issue raw messages (meant for UC DSL mode in Emacs)
   3. -margin <n> Set word wrap column to <n> (default is 78)
   4. {-help, --help} Display this list of options
   5. -units Require grouping definitions into units
      1. A *unit* is a triple consisting of a real functionality, an ideal functionality and a simulator that all fit together.
      2. This option adds a few consistency checks among these pieces, including that the basic adversarial interface implemented by the ideal functionality is not a subinterface of the composite adversarial interface implemented by the real functionality.
      3. An ideal functionality on its own is considered a singleton unit. An example is Forwarding.Forw.
3. The type checker emits no output unless there are errors to report. Errors are written to standard error output.
4. The exit status is 0 if no errors are found and 0 otherwise.

## Running the UC DSL Interpreter

UC DSL interpreter scripts are stored in text files with the ".uci" file type/suffix. File names are not otherwise restricted. The interpreter is invoked with the ucdsl command with the syntax described above. The following command-line options are available when using the interpreter:

1. {-include, -I} <dir> Add directory <dir> to the search path for theories
2. -raw-msg Issue raw messages (meant for UC DSL mode in Emacs)
3. {-help, --help} Display this list of options
4. -interpreter Run the interpreter; this is implicit if a “.uci” file is specified; omit file to  
    run the interpreter interactively
5. -batch Run interpreter on a file in batch mode (no output unless an error is  
    encountered)
6. -debug Print interpreter debugging messages

## Running in Proof General

Instructions for configuring Proof General in Emacs are in the uc-dsl/README.md file. Mode ucdsl-mode is used for “.uc” files and mode ucdsl-interpreter-mode is used for “.uci” files.

# Using EasyCrypt Theories

UC DSL files may use definitions in other UC DSL files with a statement of the form

uc\_requires <UC DSL theory name> ... .

Definitions from other files are referenced using qualified names similar to EasyCrypt's. For example, the file SMC.uc (in the EasyUC distribution under uc-dsl/examples/smc-case-study) contains

uc\_requires KeyExchange Forwarding.

each in its own .uc file. SMC.uc can now refer to KeyExchange.KEReal and Forwarding.Forw.

UC DSL files may also use types, operators and predicates defined in EasyCrypt theory files (with the ".ec" suffix). Theories are made available with a statement of the form

ec\_requires [+]<EasyCrypt theory name> ... .

The optional “+” before a theory name means to import it in addition to requiring it, meaning names do not have to be qualified unless the unqualified name is ambiguous.

The search path for ".uc" and ".ec" files consists of the current directory, the directories added to the search path with the -I or ‑include command-line options and the EasyCrypt theory directory.

Note:

* The uc\_requires line must come before the ec\_requires line and both must come before any definitions. Both are optional.
* The uc\_requires and ec\_requires directives are transitive. That is, requiring a .ec or .uc file makes the definitions of that file available, plus those of all the .ec and .uc files required (directly or indirectly) by that file.
* The import feature of ec\_requires is not transitive. That is, unqualified names can only be used for definitions in the files explicitly mentioned that have "+". Subtheories cannot be imported.

# Interfaces

Messages that are passed between the components of a UC DSL model are defined in *interfaces*. There are two kinds of interface based on the roles they play: *direct* and *adversarial*. A direct interface defines a set of message types exchanged between a functionality and the environment. These message types correspond to the operations that the functionality fulfills on behalf of entities in the environment (i.e., "principals," such as the venerable Alice and Bob). For example, a simple message forwarding functionality might use this interface:

direct FwDir' {

in pt1@fw\_req(pt2 : port, u : univ)

out fw\_rsp(pt1 : port, u : univ)@pt2

}

This interface allows one entity, abstractly represented as pt1, to transfer a value u to another entity, represented as pt2. The intended operation is as follows: the sender (i.e., the environment) sends a fw\_req message to the forwarding functionality, which contains a reference to the recipient and the value to transfer. At some later time, the functionality sends a fw\_rsp message to the recipient (again, the environment), which contains a reference to the sender and the value.

A direct interface has a name that must start with an uppercase letter, FwDir' in this case. (Names [identifiers] in UC DSL consist of letters, digits, apostrophes and underscores, with some restrictions; see Lexical Notes for details.) It contains a list of *message types*. Each message type has

* A *direction*, either in or out, from the perspective of the functionality (in from the environment or out to it)
* A name, which must start with a lowercase letter
* Optionally, a list of parameters (name : type pairs) separated by commas and enclosed in parentheses; if there are no parameters, the parentheses may be omitted. Parameter names must start with a lowercase letter.
* A *port* name. If the direction is in, the port identifies the source of the message and is put before the message name, as pt@. If the direction is out, the port identifies the recipient of the message and is put after the message name (and any parameters) as @pt. Port names must start with a lowercase letter.

A parameter type may be any valid EasyCrypt type expression. The port and univ types are built into UC DSL. A port can be thought of as an address in some abstract address space. The univ or universal type can represent any value of any type by means of explicit *encoding and partial decoding pairs* (EPDP) that convert values to and from other types. EasyUC uses the univ type for all messages and the UC DSL compiler generates code automatically for EPDPs and conversions of message arguments. In this case, however, one of the parameter types is in fact univ, exposing this bit of infrastructure.

An adversarial interface defines a set of message types exchanged between a functionality and an adversary. The effect is to define what powers the adversary has to observe and/or interfere with the functionality as it carries out its direct duties. For example, the adversarial interface for the forwarding functionality is defined as

adversarial FwAdv {

out fw\_obs(pt1 : port, pt2 : port, u : univ)

in fw\_ok

}

This interface allows the adversary to observe everything but touch nothing--it cannot reroute a message to a different destination port, spoof the source port or alter the value sent. It can also delay the delivery of a message indefinitely and hence alter the order of delivery.

The syntax of an adversarial interface is the same as that of a direct one, except for substituting the keyword adversarial for direct and omitting ports (because we know the two participants are the functionality and the adversary). The direction is again specified with respect to the functionality, out to the adversary and in from it.

Structurally, the two interfaces above are called *basic*. As we'll see later, functionalities sometimes implement more than one direct and/or adversarial interface. Wherever this is possible, a *composite* interface is used, which is simply a list of subinterfaces using basic interface names as their "types." The direct interface, in particular, is always composite, so even though the forwarding functionality only implements one basic interface, we define

direct FwDir {

D : FwDir'

}

A direct or adversarial composite interface may only contain direct or adversarial basic interfaces, respectively.

# Ideal Functionalities

An ideal functionality is a state machine that *implements* a composite direct interface and a basic adversarial interface by sending and receiving the message types they define. For example, the forwarding functionality Forw is defined as

functionality Forw implements FwDir FwAdv {

initial state Init {

match message with

| pt1@FwDir.D.fw\_req(pt2, u) => {

if (envport pt2) {

send FwAdv.fw\_obs(pt1, pt2, u)

and transition Wait(pt1, pt2, u).

}

else { fail. }

}

end

}

state Wait(pt1 : port, pt2 : port, u : univ) {

match message with

| FwAdv.fw\_ok => {

send FwDir.D.fw\_rsp(pt1, u)@pt2

and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

where the first line specifies its name and identifies the interfaces it implements. The name must start with an uppercase letter. The body of the functionality is a list of state definitions separated by white space and enclosed in braces. Each state definition has a name, which must start with an uppercase letter, and an optional list of parameters (name : type pairs) separated by commas and enclosed in parentheses; if there are no parameters, the parentheses may be omitted. Parameter names must start with a lowercase letter. The body of a state definition is an optional list of local variable declarations followed by a match statement to be explained shortly.

A functionality has an implicit state variable whose value is the current state along with its parameters. The functionality starts in the state marked initial; this state must not have parameters.

The state machine operates as a Mealy machine: given an input message and the current state, it produces an output message and transitions to a new state (which might be the same state). In terms of control flow, a state machine is passive until a message is received, at which time it performs some computation, generates an output message and records its next state. Control is then transferred to the recipient of the output message. Should control ever return (via receipt of a new message), the machine resumes in the new state.

The body of a state definition defines how the functionality responds to messages when in that state. The input message is stored in a built-in variable named message. A state can also define local variables using the syntax

var v : t;

where v is an identifier that must start with a lowercase letter and t is an EasyCrypt type expression. The variable is not referenceable outside the state and its value is undefined upon each entry to the state (it is not persistent). Its name must be distinct from those of the state parameters.

The remainder of the body is a match statement that is used to determine which message was received and how to respond. The general syntax is

match message with

pattern1 => { statements1 }

| ...

| patternN => { statementsN }

end

where pattern1 may optionally be preceded by | as well. A pattern consists of a *message path* applied to a *pattern variable* for each message parameter and the source port (if the message is in a direct interface). The input message is matched against each pattern in turn until the first successful match is found that instantiates all the pattern variables. For example, the pattern

pt1@FwDir.D.fw\_req(pt2, u)

in the Init state above matches any fw\_req message as defined in the D subinterface of the composite direct interface FwDir. A successful match binds the pattern variable pt1 to the source port of the incoming message and pt2 and u to the data (or arguments) of the message. Pattern variable names are arbitrary and need not match those used in the interface definition, but must not duplicate the names of any state parameters or local variables.

The only other message this functionality can receive is matched by the pattern

FwAdv.fw\_ok

which is matched in the Wait state above. This is an adversarial message, so there is no source port, and this one has no data, either.

As in EasyCrypt, if one or more arguments of the message are not relevant to the action to be taken, the special pattern variable \_ may be used to show they are ignored:

pt1@FwDir.D.fw\_req(\_, \_)

The source port, however, cannot be replaced with \_.

Finally, a wild card pattern, \*, can be used to replace the trailing part of the message path. When \* is used, pattern variables for the source port and arguments are omitted. For example,

FwDir.D.\*

FwDir.\*

FwAdv.\*

\*

are all valid patterns. The only statement that can be associated with a wild card pattern is unconditional failure, as explained below.

All possible in message paths, as defined by the interfaces implemented by the ideal functionality, must be matched by at least one pattern in the match statement, except that the initial state cannot receive adversarial messages. Each pattern must also be *reachable*, meaning at least one message that matches the pattern does not match any previous pattern (e.g., no duplicate patterns). In the example above, the match statement for the Init state has a pattern for the direct message only, the one for the Wait state has two patterns, the second being a wildcard, and the one for the Final state has one wildcard pattern matching both possible messages.

Each pattern has an associated sequence of statements that is executed when the pattern is matched. Statements may use the values of the state's parameters and the message parameters but may not modify them. They may also use or set the values of local variables. Statement types include a subset of those provided by EasyCrypt for procedure bodies:

1. Assignment of a value (i.e., an EasyCrypt typed expression) to a variable, tuple or map key

x <- 0;

(x, y) <- (y, x);

m.[x] <- y;

Note: As in EasyCrypt, you cannot assign a value to a field of a tuple or record:

x.`1 <- 15; (\* Not legal \*)

This is because .` is an operator, so x.`1 is an expression rather than a variable.

1. Assignment of a value sampled from a (sub-)distribution to a variable, tuple or map key

x <$ [5..10];

1. A match statement

match exp with

pattern1 => { statements1 }

| ...

| patternN => { statementsN }

end

where exp is an EasyCrypt typed expression whose value belongs to an inductive and each pattern is an EasyCrypt pattern consisting of a constructor of the type applied to pattern variables. These variables may only be used in the corresponding statement block. There must be one pattern-statement pair for each constructor of the type. A | may optionally precede pattern1.

1. A conditional statement

if (b) { <statements> }

[ elif (bi) { <statements> } ]

[ else { <statements> } ]

where b and bi are EasyCrypt typed expressions of type bool. Any number of elif branches can be used. Unlike in EasyCrypt, braces are required even if a branch consists of a single statement.

Notably absent are procedure calls (because EasyCrypt modules are stateful and the only notion of state in EasyUC is provided by state machines), while loops and return statements.

The last statement on every execution path must be one of two special statements terminated with a period:

* Send a message and transition to a new state:

send M and transition S.

where M is a message path and any associated arguments and destination port and S is a *non-initial* state and any associated arguments. For example,

send FwAdv.fw\_obs(pt1, pt2, u)

and transition Wait(pt1, pt2, u).

sends a message to the adversary and transitions to the Wait state. In an ideal functionality, the initial state can send *only* adversarial messages. Other states, of course, can send direct messages, such as

send FwDir.D.fw\_rsp(pt1, u)@pt2

and transition Final.

Here the destination port is the identifier pt2. If the port is an expression that is not just an identifier, it must be in parentheses:

send FwDir.D.fw\_rsp(pt1, u)@(if b then pt2 else pt3)

and transition Final.

* Unconditionally fail, which gives control back to the "root" of the environment without a message and without changing the current state:

fail.

Figure 1 depicts the state machine of Forw. Direct messages are shown in black, adversarial ones in red.

A picture containing graphical user interface

Description automatically generated

Figure 1 Forw State Machine

The operation of Forw can be summarized as follows:

* In the initial state, Init, it waits to receive an fw\_req message containing pt2 and u. When it does, pt1 is guaranteed to point into the environment (i.e., not into the forwarder or adversary) because it came via a direct interface. It checks (with the built-in envport predicate) that pt2 also points into the environment and fails if it doesn't (i.e., the request is invalid). If it does, it "leaks" the forwarding request by sending an fw\_obs message containing pt1, pt2 and u to the adversary and enters the Wait state, recording pt1, pt2 and u.
* In the Wait state, it waits to receive an fw\_ok message from the adversary to allow the forwarding. When it does, it sends a fw\_rsp message containing pt1 and u to pt2 and enters the Final state.
* In the Final state, it fails if any message is received.

# Real Functionalities

A real functionality is a collection of *parties*. The name of a party must start with an uppercase letter. A party is a state machine that optionally *serves* a basic direct interface and/or a basic adversarial interface by sending and receiving the message types they define. The real functionality as a whole implements a composite direct interface containing all of the parties' basic ones and an optional composite adversarial one that does likewise. Each basic direct interface in the composite must be served by exactly one party.

A real functionality may also contain *subfunctionalities*, which are instances (copies) of ideal functionalities. An example of a subfunctionality declaration is

subfun Fw1 = Forwarding.Forw

The name of a subfunctionality (such as Fw1) must start with an uppercase letter and the name of the functionality of which it is a copy must be qualified (preceded by the name of its containing theory). Parties do not typically send or receive messages among themselves, though nothing prevents this. Instead, each party P uses its *internal port*, denoted intport P, to send and receive messages to and from the subfunctionalities. The real functionality as a whole does not implement the interfaces implemented by its subfunctionalities, because the environment is not allowed to send or receive messages to or from them; however, message match statements in the states of parties, including initial states, must include patterns matching all possible direct out message paths from subfunctionalities. Likewise, send-and-transition statements may use any direct in message path of a subfunctionality, even in initial states.

For example, KEReal is a real functionality that allows two entities in the environment to establish a shared secret using Diffie-Hellman key exchange. The parties of the functionality act as proxies for these entities to carry out the exchange. They communicate with each other over an untrusted network by means of two subfunctionalities, Fw1 and Fw2, that are instances of the Forw functionality above. Each party randomly selects a value used for the key exchange.

Here are the theories that KEReal requires

uc\_requires Forwarding.

ec\_requires +KeysExponentsAndPlaintexts.

The file Forwarding.uc contains the Forw ideal functionality defined in the previous section. The file KeysExponents­And­Plain­texts.ec contains an EasyCrypt "support" theory that defines types and operators used by the functionalities. Elements of Diffie-Hellman key exchange include a key type with the operators and axioms of a group, an exp type with the operators and axioms of a commutative semi-group, the uniform, full and lossless distribution on exp, and a key value named g that uniquely generates every key by exponentiation with a value of exp. Other operators include EPDPs for converting among various types, such as key and univ, including types for values to be sent in messages, such as (port \* port \* key) and univ.

Here are the interfaces KEReal implements.

direct KEDirPt1 { (\* Party 1 \*)

in pt1@ke\_req1(pt2 : port)

out ke\_rsp2(k : key)@pt1

}

direct KEDirPt2 { (\* Party 2 \*)

out ke\_rsp1(pt1 : port, k : key)@pt2

in pt2@ke\_req2

}

direct KEDir {

Pt1 : KEDirPt1 (\* Party 1 \*)

Pt2 : KEDirPt2 (\* Party 2 \*)

}

The composite direct interface KEDir consists of two basic direct interfaces, one for each party. The parties leak no information, so there are no adversarial interfaces. The Forw subfunctionalities leak information to the adversary "under the covers," as it were.

Here is the definition of KEReal:

functionality KEReal implements KEDir {

subfun Fw1 = Forwarding.Forw

subfun Fw2 = Forwarding.Forw

party Pt1 serves KEDir.Pt1 {

initial state WaitReq1 {

var q1 : exp;

match message with

| pt1@KEDir.Pt1.ke\_req1(pt2) => {

if (envport pt2) {

q1 <$ dexp;

send Fw1.D.fw\_req

(intport Pt2,

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g^q1))

and transition WaitFwd2(pt1, q1).

}

else { fail. }

}

| \* => { fail. }

end

}

state WaitFwd2(pt1 : port, q1 : exp) {

match message with

| Fw2.D.fw\_rsp(\_, u) => {

match epdp\_key\_univ.`dec u with

| Some k2 => {

send KEDir.Pt1.ke\_rsp2(k2 ^ q1)@pt1

and transition Final.

}

| None => { fail. } (\* will never happen \*)

end

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

party Pt2 serves KEDir.Pt2 {

initial state WaitFwd1 {

var q2 : exp; var pt1, pt2 : port; var k1 : key;

match message with

| Fw1.D.fw\_rsp(\_, u) => {

match epdp\_port\_port\_key\_univ.`dec u with

| Some tr => {

(pt1, pt2, k1) <- tr;

q2 <$ dexp;

send KEDir.Pt2.ke\_rsp1(pt1, k1 ^ q2)@pt2

and transition WaitReq2(pt2, q2).

}

| None => { fail. } (\* cannot happen \*)

end

}

| \* => { fail. }

end

}

state WaitReq2(pt2 : port, q2 : exp) {

match message with

| pt2'@KEDir.Pt2.ke\_req2 => {

if (pt2' = pt2) {

send Fw2.D.fw\_req(intport Pt1,

epdp\_key\_univ.`enc (g^q2))

and transition Final.

}

else { fail. }

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

}

Figure 2 depicts the state machines of the KEReal parties. The intport operator is abbreviated iport and EPDP encoding operators are abbreviated enc.

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

Figure 2 State Machines of the KEReal Parties Pt1 (top) and Pt2 (bottom)

Figure 3 depicts the KEReal message flow as a UML sequence diagram[[1]](#footnote-2).

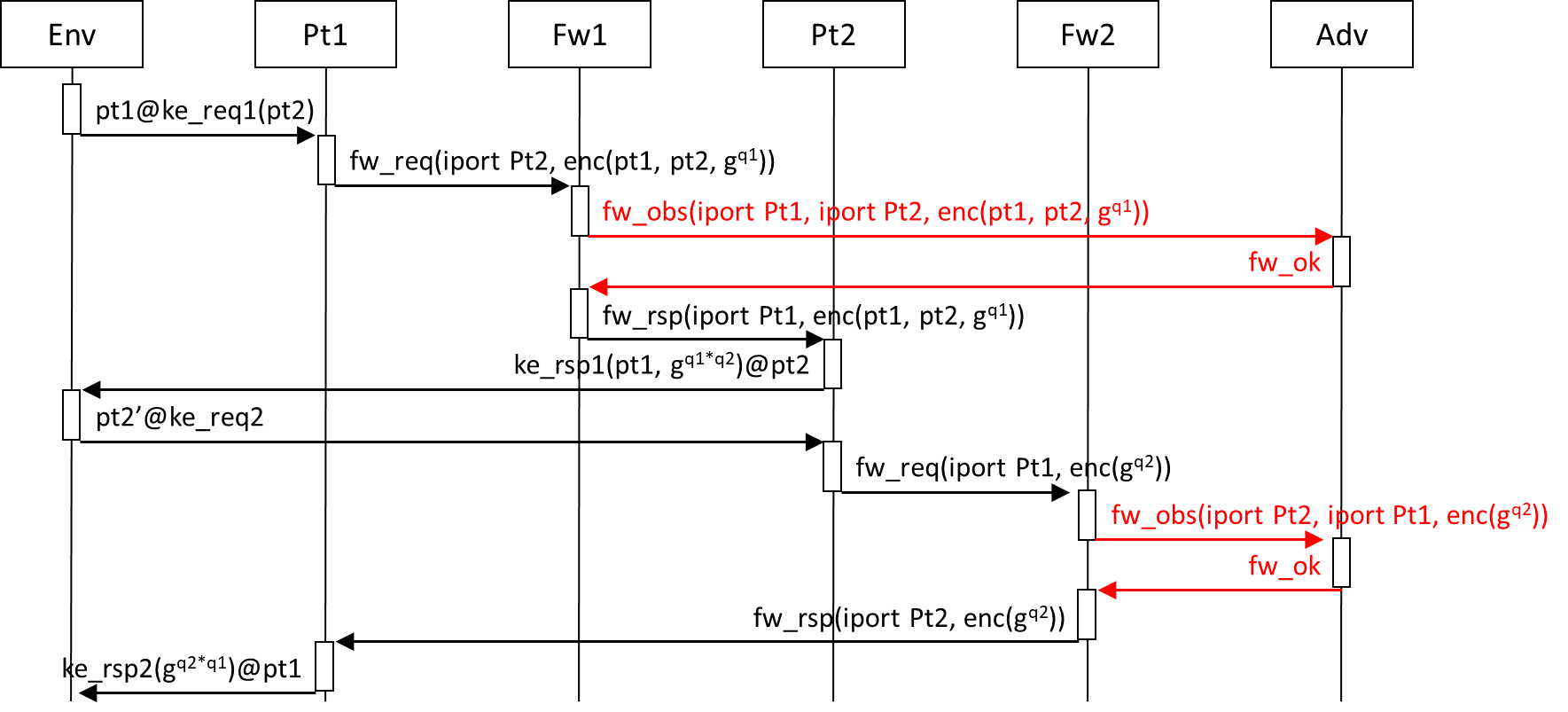


Figure 3 KEReal Message Flow

Upon completion, both entities in the environment know the combined key, gq1\*q2 = gq2\*q1 and the adversary knows gq1 and gq2. The security of the exchange relies on the fact that determining the combined key from these two pieces is impossible within a feasible amount of computation (known as the Diffie-Hellman assumption).

**Note**. To a subfunctionality, the parties of the parent functionality are part of the environment—they are external to it and not part of the adversary. Thus "environment" is a relative term. For example, when Pt1 sends a fw\_req message to Fw1 with intport Pt2 as its first argument, Fw1 checks that the argument is an envport. It is, because it is external to Fw1: it is in the environment *of Fw1* even though it is not in "the" environment.

# Simulators

A simulator is an ideal adversary that translates between an ideal functionality and a real functionality's adversary in order to show the real functionality UC-realizes the ideal functionality. That is, the job of a simulator is to make an ideal functionality and a real functionality indistinguishable to the adversary. A simulator is a state machine. A simulator deals only with adversarial messages (and message paths).

For example, KEIdeal is an ideal functionality that models the essence of a key exchange without worrying about details such as untrustworthy networks. It implements the same composite direct interface as KEReal and this basic adversarial interface:

adversarial KEI2S {

out ke\_sim\_req1(pt1 : port, pt2 : port)

in ke\_sim\_rsp

out ke\_sim\_req2

}

Here is the definition of KEIdeal:

functionality KEIdeal implements KEDir KEI2S {

initial state WaitReq1 {

match message with

| pt1@KEDir.Pt1.ke\_req1(pt2) => {

if (envport pt2) {

send KEI2S.ke\_sim\_req1(pt1, pt2)

and transition WaitSim1(pt1, pt2).

}

else { fail. }

}

| \* => { fail. }

end

}

state WaitSim1(pt1 : port, pt2 : port) {

var q : exp;

match message with

| KEI2S.ke\_sim\_rsp => {

q <$ dexp;

send KEDir.Pt2.ke\_rsp1(pt1, g ^ q)@pt2

and transition WaitReq2(pt1, pt2, q).

}

| \* => { fail. }

end

}

state WaitReq2(pt1 : port, pt2 : port, q : exp) {

match message with

| pt2'@KEDir.Pt2.ke\_req2 => {

if (pt2' = pt2) {

send KEI2S.ke\_sim\_req2

and transition WaitSim2(pt1, pt2, q).

}

else { fail. }

}

| \* => { fail. }

end

}

state WaitSim2(pt1 : port, pt2 : port, q : exp) {

match message with

| KEI2S.ke\_sim\_rsp => {

send KEDir.Pt1.ke\_rsp2(g ^ q)@pt1 and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

Figure 4 depicts the state machine of KEIdeal.

Diagram

Description automatically generated with medium confidence

Figure 4 KEIdeal State Machine

In short, KEIDeal generates one value, shares it with pt1 and pt2 and reveals to the adversary only the ports that now share the value. The security of this arrangement is self-evident.

KESim is a simulator that makes KEIDeal and KEReal indistinguishable to the adversary. It translates between the adversarial interface of KEIDeal (namely KEI2S) and the adversarial interfaces of KEReal (namely of its Forw subfunctionalities). KESim *uses* KEI2S, which means it acts as the adversary of KEIDeal: out messages of KEI2S are *to* KESim and in messages are *from* it, because the directions are with respect to KEIDeal. It *simulates* KEReal, which determines the adversarial interfaces it implements, including those of its subfunctionalities (and parameters, which will be discussed in the next section).

Here is the definition of KESim:

simulator KESim uses KEI2S simulates KEReal {

initial state WaitReq1 {

var q1 : exp;

match message with

| KEI2S.ke\_sim\_req1(pt1, pt2) => {

q1 <$ dexp;

send KEReal.Fw1.FwAdv.fw\_obs

(intport KEReal.Pt1, intport KEReal.Pt2,

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ q1))

and transition WaitAdv1(q1).

}

| \* => { fail. }

end

}

state WaitAdv1(q1 : exp) {

var q2 : exp;

match message with

| KEReal.Fw1.FwAdv.fw\_ok => {

q2 <$ dexp;

send KEI2S.ke\_sim\_rsp and transition WaitReq2(q1, q2).

}

| \* => { fail. }

end

}

state WaitReq2(q1 : exp, q2 : exp) {

match message with

| KEI2S.ke\_sim\_req2 => {

send KEReal.Fw2.FwAdv.fw\_obs

(intport KEReal.Pt2, intport KEReal.Pt1,

epdp\_key\_univ.`enc (g ^ q2))

and transition WaitAdv2(q1, q2).

}

| \* => { fail. }

end

}

state WaitAdv2(q1 : exp, q2 : exp) {

match message with

| KEReal.Fw2.FwAdv.fw\_ok => {

send KEI2S.ke\_sim\_rsp and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

Figure 5 depicts the state machine of KESim. In short, KESim generates two values that it leaks to the adversary that *have nothing to do with* the value generated by KEIdeal to be shared by pt1 and pt2. The various ports also leaked to the adversary are identical to those that would be sent by KEReal. Again, the indistinguishability of this simulation depends on the adversary's inability to relate the values leaked to it to the shared key, to which in fact they are not related. [2] provides a formal proof of this claim in EasyCrypt.

Timeline

Description automatically generated

Figure 5 KESim State Machine

Figure 6 depicts the message flow through KEIdeal and KESim. Comparing it to Figure 3, it is apparent that the flows into and out of Env and Adv are identical except for Env receiving the shared value gq rather than gq1\*q2.

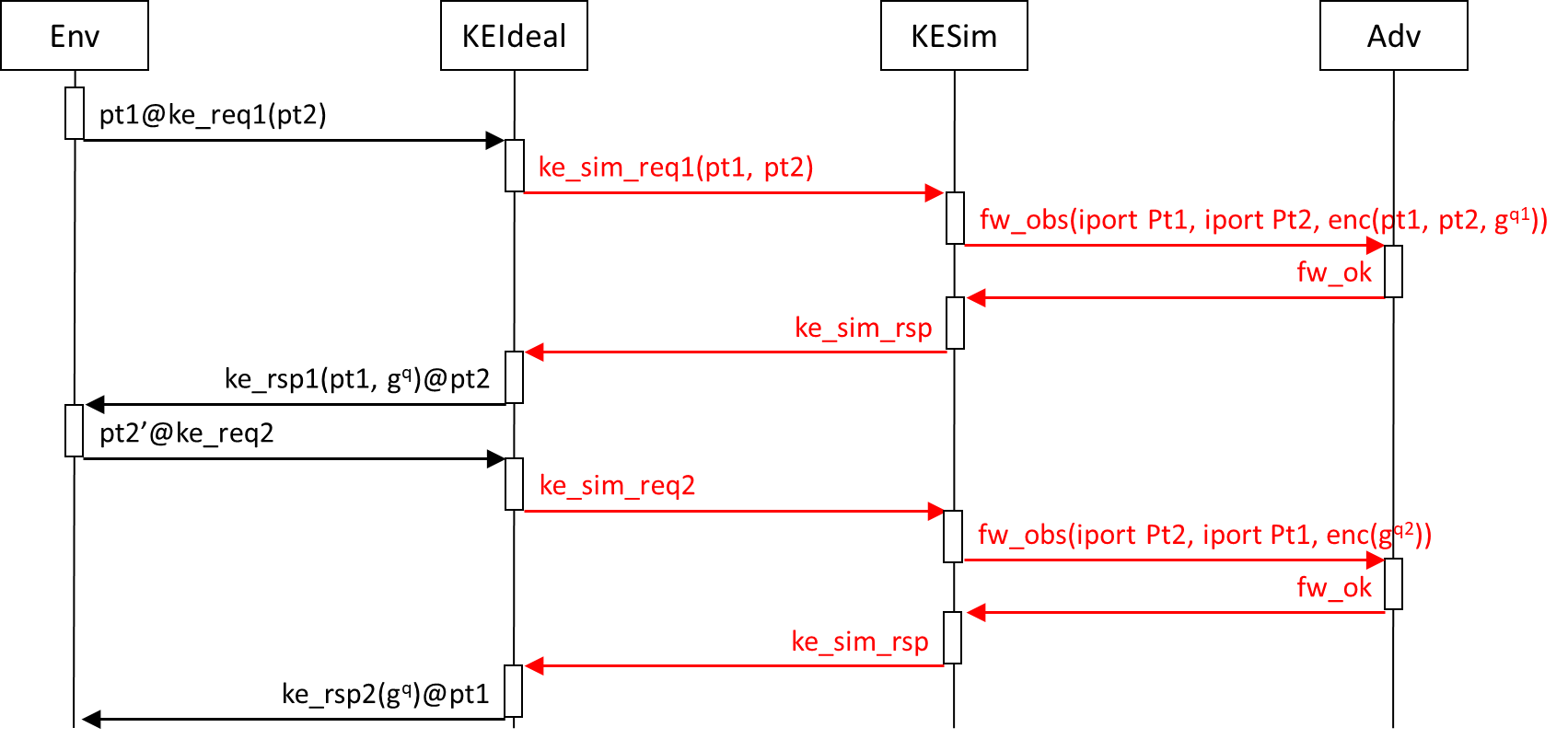


Figure 6 KEIdeal and KESim Message Flow

# Composing Functionalities

Canetti [1] defines the *UC operation* as taking a real functionality ρ that uses an ideal functionality φ and a real functionality π that is compatible with φ and returning the protocol ρφ→π in which each instance of φ, when called as a subroutine of ρ, is replaced by an instance of π. The UC Theorem proves that if π uc-realizes φ then the UC operation preserves security. That is, if ρ uc-realizes some ideal functionality ι, then so does ρφ→π.

UC DSL represents composition of functionalities by allowing a real functionality to be parameterized on one or more ideal functionalities. This is similar to using an ideal functionality as a subfunctionality, except subfunctionalities are internal and not visible to the outside world, whereas parameters are part of the public interface of the real functionality (though the environment still can't communicate with them directly).

For example, Secure Message Communication (SMC) models the case where one entity wishes to send a confidential value to another: essentially what Forw does but without revealing the value to the adversary. The theory SMC is a *unit* consisting of real and ideal functionalities, SMCReal and SMCIdeal, respectively, their associated interfaces and a simulator, SMCSim. The environment interacts with the functionalities through ports pt1 and pt2. The communication is initiated from pt1 and the value (or *text*) is received by pt2. The composite direct interface has two subinterfaces, corresponding to the two parties of the real functionality:

direct SMCPt1 {

in pt1@smc\_req(pt2 : port, t : text)

}

direct SMCPt2 {

out smc\_rsp(pt1 : port, t : text)@pt2

}

direct SMCDir {

Pt1 : SMCPt1

Pt2 : SMCPt2

}

The ideal functionality has this adversarial interface, which reveals only the ports involved in the communication, not the value communicated:

adversarial SMC2Sim {

out sim\_req(pt1 : port, pt2 : port)

in sim\_rsp

}

Here is the definition of SMCIdeal:

functionality SMCIdeal implements SMCDir SMC2Sim {

initial state WaitReq {

match message with

| pt1@SMCDir.Pt1.smc\_req(pt2, t) => {

if (envport pt2) {

send SMC2Sim.sim\_req(pt1, pt2)

and transition WaitSim(pt1, pt2, t).

}

else { fail. }

}

| \* => { fail. }

end

}

state WaitSim(pt1 : port, pt2 : port, t : text) {

match message with

| SMC2Sim.sim\_rsp => {

send SMCDir.Pt2.smc\_rsp(pt1, t)@pt2 and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

Figure 7 depicts the state machine of SMCIdeal.

Graphical user interface

Description automatically generated with medium confidence

Figure 7 SMCIdeal State Machine

In short, SMCIdeal receives a text t from pt1 and delivers it to pt2, revealing to the adversary only the ports pt1 and pt2. It is self-evident that this functionality does not reveal t.

The real functionality carries out the communication by encrypting the text using a one-time pad agreed using a key exchange. SMCReal has two parties and one Forw subfunctionality. In addition, SMCReal is *composed* with a key exchange functionality by parameterizing it on a functionality that implements the composite direct interface KeyExchange.KEDir. This parameter will be instantiated later with KEIdeal. Parties use a parameter in the same way as they use a subfunctionality.

Here is the definition of SMCReal:

functionality SMCReal(KE : KeyExchange.KEDir) implements SMCDir {

subfun Fwd = Forwarding.Forw

party Pt1 serves SMCDir.Pt1 {

initial state WaitReq {

match message with

| pt1@SMCDir.Pt1.smc\_req(pt2, t) => {

if (envport pt2) {

send KE.Pt1.ke\_req1(intport Pt2)

and transition WaitKE2(pt1, pt2, t).

}

else { fail. }

}

| \* => { fail. }

end

}

state WaitKE2(pt1 : port, pt2 : port, t : text) {

match message with

| KE.Pt1.ke\_rsp2(k) => {

send Fwd.D.fw\_req

(intport Pt2,

epdp\_port\_port\_key\_univ.`enc

(pt1, pt2, epdp\_text\_key.`enc t ^^ k))

and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

party Pt2 serves SMCDir.Pt2 {

initial state WaitKE1 {

match message with

| KE.Pt2.ke\_rsp1 (\_, k) => {

send KE.Pt2.ke\_req2 and transition WaitFwd(k).

}

| \* => { fail. }

end

}

state WaitFwd(k : key) {

var pt1, pt2 : port; var x : key;

match message with

| Fwd.D.fw\_rsp(\_, u) => {

match epdp\_port\_port\_key\_univ.`dec u with

| Some tr => {

(pt1, pt2, x) <- tr;

match epdp\_text\_key.`dec (x ^^ kinv k) with

| Some t => {

send SMCDir.Pt2.smc\_rsp(pt1, t)@pt2

and transition Final.

}

| None => { fail. } (\* cannot happen \*)

end

}

| None => { fail. } (\* cannot happen \*)

end

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

}

Figure 8 depicts the state machines of the parties of SMCReal.

Graphical user interface, application

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Figure 8 State Machines of the SMCReal Parties Pt1 (top) and Pt2 (bottom)

Figure 9 depicts the message flow through SMCReal. It treats the parameter KE as a black box, hiding its internal communications.

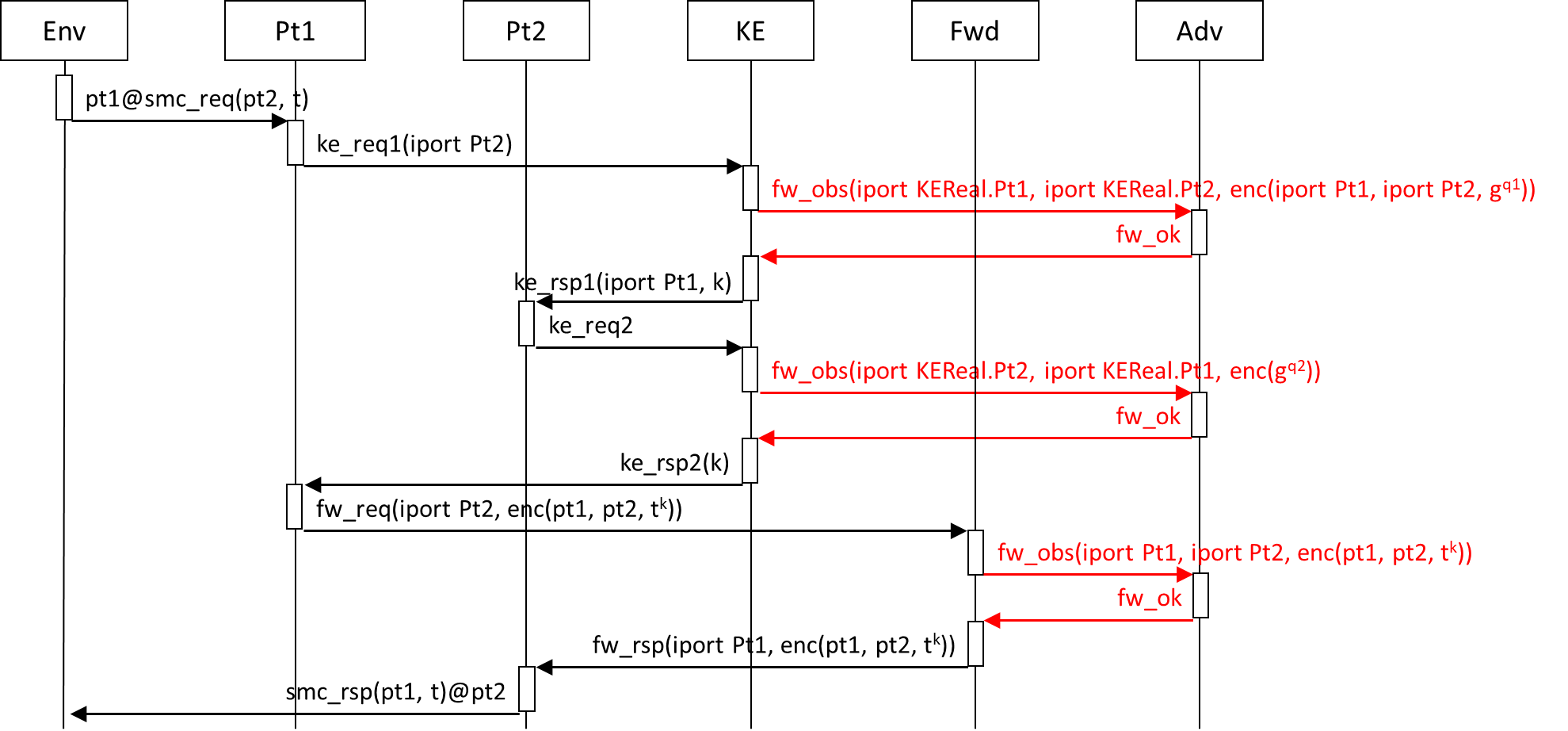


Figure 9 SMCReal(KE) Message Flow

The simulator, SMCSim, provides the information needed to prove that leaking tk does not compromise the security of SMCReal by leaking an unrelated value that the adversary can only distinguish from the real thing with negligible probability. Because SMCReal has a parameter, the definition of SMCSim must provide an argument. Using the notation introduced at the start of this section, SMCReal is ρ, KEIdeal is φ, KEReal is π and SMCIdeal is ι. Thus SMCReal(KEIdeal) corresponds to ρφ (abusing the notation a bit) and SMCSim will be used to prove that SMCReal(KEIdeal) uc-realizes SMCIdeal, if we assume (or have proved) that KEReal uc-realizes KEIdeal. On a practical level, the argument to SMCReal also informs the simulator which adversarial interface it must implement in addition to FwAdv, namely KEI2S.

Here is the definition of SMCSim:

simulator SMCSim uses SMC2Sim

simulates SMCReal(KeyExchange.KEIdeal) {

initial state WaitReq {

match message with

| SMC2Sim.sim\_req(pt1, pt2) => {

send SMCReal.KE.KEI2S.ke\_sim\_req1

(intport SMCReal.Pt1, intport SMCReal.Pt2)

and transition WaitAdv1(pt1, pt2).

}

end

}

state WaitAdv1(pt1 : port, pt2 : port) {

var q : exp;

match message with

| SMCReal.KE.KEI2S.ke\_sim\_rsp => {

q <$ dexp;

send SMCReal.KE.KEI2S.ke\_sim\_req2

and transition WaitAdv2(pt1, pt2, q).

}

| \* => { fail. }

end

}

state WaitAdv2(pt1 : port, pt2 : port, q : exp) {

match message with

| SMCReal.KE.KEI2S.ke\_sim\_rsp => {

send SMCReal.Fwd.FwAdv.fw\_obs

(intport SMCReal.Pt1, intport SMCReal.Pt2,

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ q))

and transition WaitAdv3(pt1, pt2, q).

}

| \* => { fail. }

end

}

state WaitAdv3(pt1 : port, pt2 : port, q : exp) {

match message with

| SMCReal.Fwd.FwAdv.fw\_ok => {

send SMC2Sim.sim\_rsp and transition Final.

}

| \* => { fail. }

end

}

state Final {

match message with

| \* => { fail. }

end

}

}

Figure 10 depicts the state machine of SMCSim.

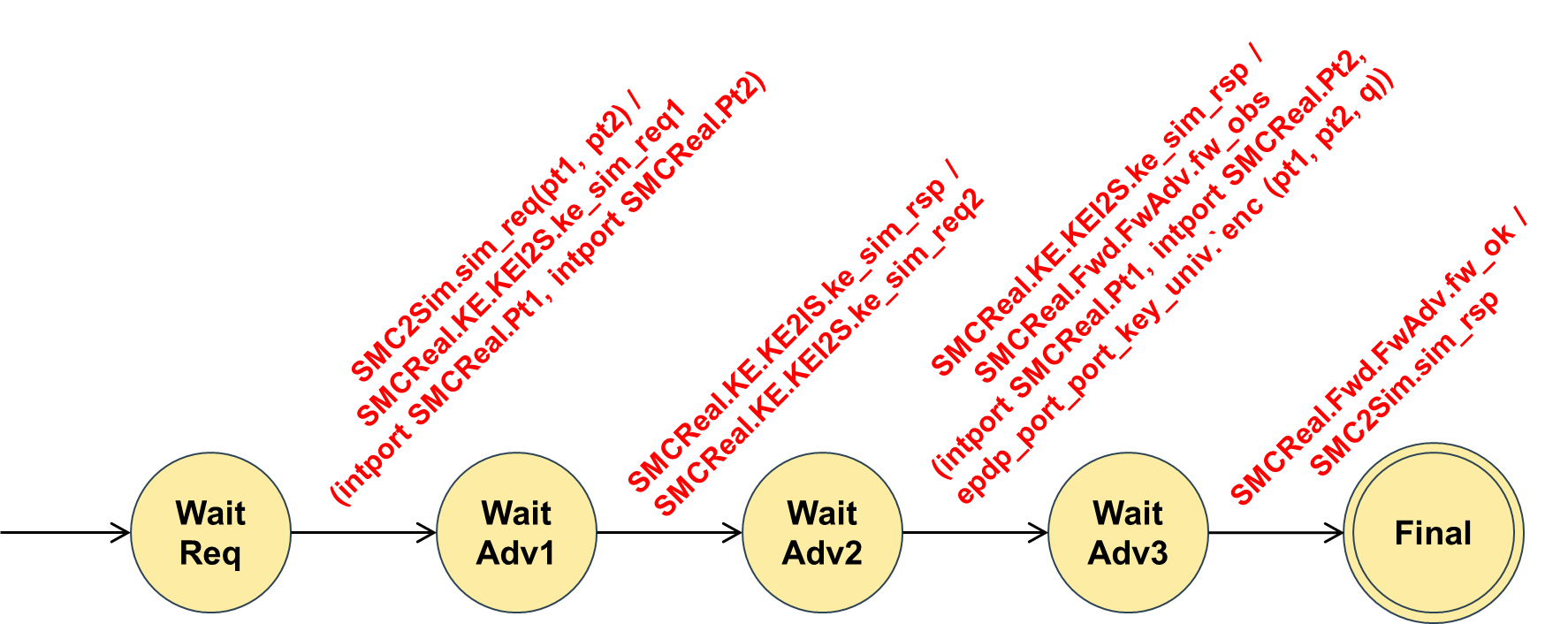


Figure 10 SMCSim State Machine

In short, SMCSim generates a value and leaks it to the adversary that *has nothing to do with* the text SimIdeal receives from pt1 and sends to pt2. The nested simulator KESim also leaks messages to simulate a key exchange. The indistinguishability of this simulation depends on the adversary's inability to distinguish three unrelated randomly generated values from a valid key exchange and transmission of an encrypted text. Again, [2] provides a formal proof of this claim in EasyCrypt.

Figure 11 depicts the message flow through SMCIdeal and SMCSim. Compare it to Figure 9.

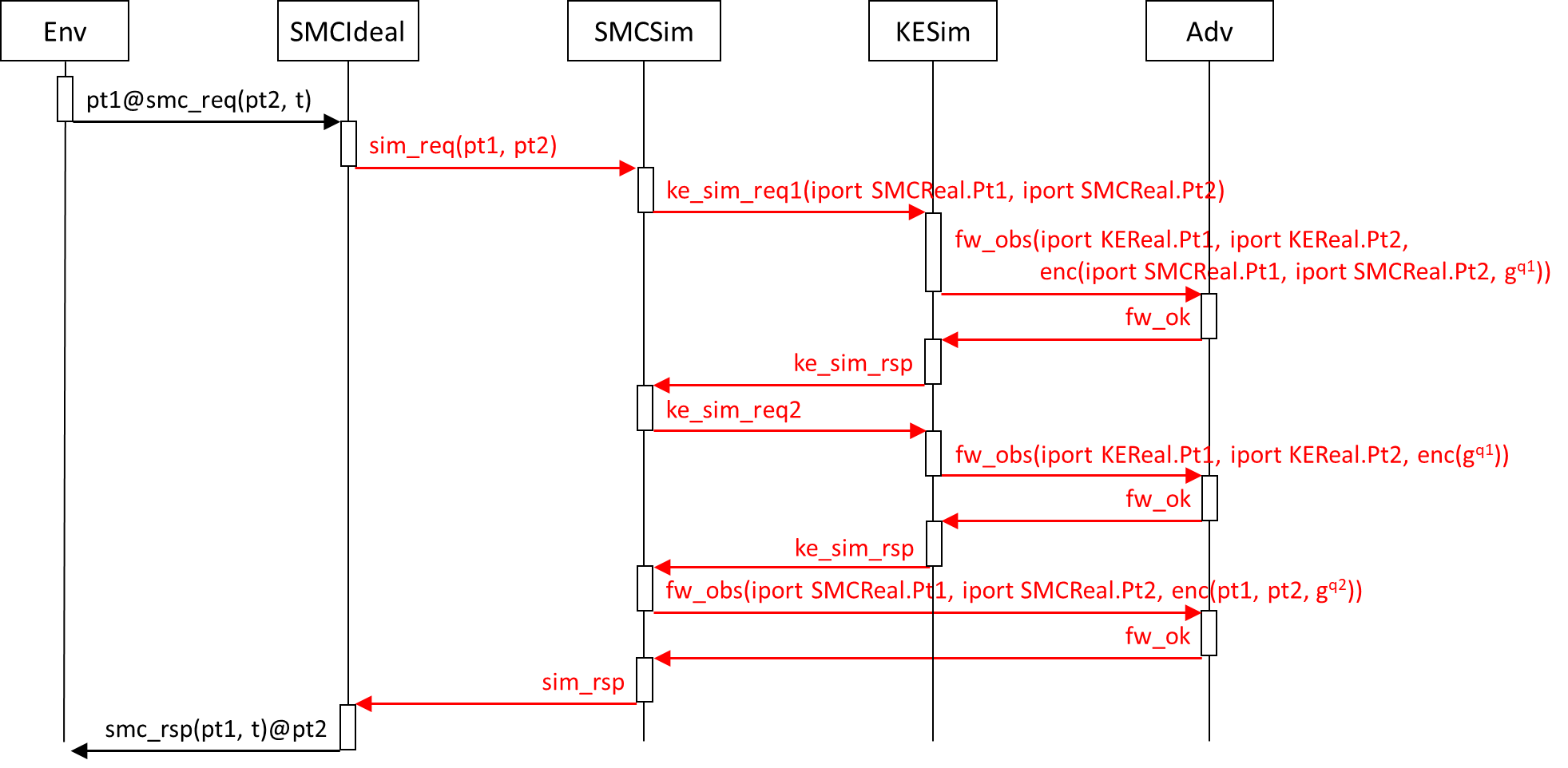


Figure 11 SMCIdeal and SMCSim Message Flow

# The UC DSL Interpreter

The purpose of the UC DSL interpreter is to execute a model, such as to test, debug or demonstrate its behavior. The interpreter script acts as the environment and adversary, sending messages to the model and receiving messages from it. See *Running the UC DSL Interpreter* for how to invoke the interpreter on a script.

Figure 12 shows a complete script for exercising the KeyExchange theory from earlier sections. We will walk through this script one line at a time.

## Script Prelude

The first line of a script specifies a UC DSL theory (or model):

**load KeyExchange.**

uc file KeyExchange loaded

The interpreter loads the theory and verifies that it contains a ***unit***, meaning an ideal functionality on its own or a triple consisting of a real functionality, an ideal functionality and a simulator that fit together by their interfaces. If the theory is a singleton unit, however, you won’t be able to do the next step.

The second line of a script specifies the configuration of the real functionality by providing arguments for its parameters. In this case, KEReal has none but the step is still required.

**functionality KEReal.**

global context:

(func : addr, adv : addr, IncFuncAdv : inc func adv)

worlds: KeyExchange.KEReal:1 ~ KeyExchange.KEIdeal:1 / KeyExchange.KESim:1

|  |
| --- |
| load KeyExchange.  real KEReal.  var pt1 : port.  var pt2 : port.  assumption pt1\_env : envport func pt1.  assumption pt2\_env : envport func pt2.  real.  send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1)).  run.  send ((adv, 2))@Forwarding.FwAdv.fw\_ok@((func ++ [1], 1)).  run.  send pt2@KeyExchange.KEDir.Pt2.ke\_req2@((func, 2)).  run.  send ((adv, 3))@Forwarding.FwAdv.fw\_ok@((func ++ [2], 1)).  run.  assert msg\_out  ((func, 1))@KeyExchange.KEDir.Pt1.ke\_rsp2(g ^ rand1 ^ rand)@pt1  ctrl\_env.  finish.  ideal.  send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1)).  run.  send ((adv, 2))@Forwarding.FwAdv.fw\_ok@((func ++ [1], 1)).  run.  send pt2@KeyExchange.KEDir.Pt2.ke\_req2@((func, 2)).  run.  send ((adv, 3))@Forwarding.FwAdv.fw\_ok@((func ++ [2], 1)).  run.  assert msg\_out  ((func, 1))@KeyExchange.KEDir.Pt1.ke\_rsp2(g ^ rand1 ^ rand)@pt1  ctrl\_env.  finish. |

Figure 12 Interpreter Script for KeyExchange Model

Given the real configuration, the interpreter sets up and displays a ***global context***. This initially contains two variables, func and adv, which are addresses. Every functionality in the broad sense has an address: this includes ideal and real functionalities as well as the environment, adversaries and simulators. The addr type is defined in the EasyCrypt theory UCBasicTypes in EasyUC’s prelude folder as int list. The real and ideal functionalities are assigned the same address, which is the value of func (if they were different, the environment could trivially tell them apart). In particular, the address of the environment is [], func is [1] and adv is [2]. Addresses are hierarchical: for example, the subfunctionalities Fw1 and Fw2 inside KEReal are assigned addresses that start with func, namely [1; 1] and [1; 2], respectively. We say that func is *less than* these addresses because it is a prefix of them. This implies that func and adv are part of the environment, which is interesting but not germane.

The global context also contains an assumption called IncFuncAdv that inc func adv is true. The inc operator takes two addresses and returns whether they are *incomparable*: whether neither is a prefix of the other, which would indicate that one of them is an internal component of the other. The lists [1] and [2] are incomparable.

The interpreter then displays the two worlds, real and ideal, that we are claiming are equivalent (indicated by ~). The numbers after the colons will be explained later.

To give a more complex example, if we were working with the SMC theory, a script might start

load SMC.

functionality SMCReal(KeyExchange.KEIdeal).

where KeyExchange.KEIdeal is the argument for the KE parameter of SMCReal. Recall this is what the simulator SMCSim simulates. The resulting worlds would be

worlds:

SMC.SMCReal:1(KeyExchange.KEIdeal:3) ~ SMC.SMCIdeal:1 /

SMC.SMCSim:1(3)

Although not shown, the SMCReal.KEIdeal is assigned address [1; 1] and the subfunctionality SMCReal.Fwd is assigned address [1; 2].

In the UC DSL, an argument must be an ideal functionality, but the interpreter is not so picky: if we wanted, we could instead specify

functionality SMCReal(KeyExchange.KEReal).

and the resulting worlds would be

worlds:

SMC.SMCReal:1(KeyExchange.KEReal:3) ~ SMC.SMCIdeal:1 /

SMC.SMCSim:1(3) \* KeyExchange.KESim:3

Here the simulator is seen to be a composite: a pair consisting of SMCSim and KESim. In this real world, SMCReal.KEReal is assigned address [1; 1] and SMCReal.Fwd is assigned [1; 2], as before, and further, SMCReal.KEReal.Fw1 is assigned [1; 1; 1] and SMCReal.KEReal.Fw2 is assigned [1; 1; 2].

If KEReal had parameters, they would need arguments, too, and so on for as many levels as needed.

## Variables and Assumptions

At this point you can do one of four things: declare a variable, state an assumption or start an interpretation session in either the ideal or real world. Variables are added to the global context for use in other scripting commands, to represent values sent to the model in messages and those received from it. Assumptions are EasyCrypt expressions of type bool that state invariant properties of or among variables. For example, the next four lines declare two ports and assumptions that they belong to the environment. A *port*, recall, is the source or destination of a message. We met envport in *Ideal Functionalities* as a one-parameter function. The envport operator here is an internal version that takes two arguments: an address and a port. It returns whether the port’s address is outside the address and outside adv, so if pt1 and pt2 are not in func or adv, they must be in the environment. The one-argument version used in the DSL simply fills in func automatically. After each of these steps, the interpreter prints the global context and the worlds again. The worlds don’t change, but the global context is now

global context:

(func : addr, adv : addr, IncFuncAdv : inc func adv, pt1 : port,

pt2 : port, pt1\_env : envport func pt1, pt2\_env : envport

func pt2)

## Real World Sessions

An interpretation session begins with one of the commands real or ideal. The commands available in a session are the same, but details of the output differ. Figure 12 does real first and the output is

**real.**

global context:

(func : addr, adv : addr, IncFuncAdv : inc func adv, pt1 : port,

pt2 : port, pt1\_env : envport func pt1, pt2\_env : envport

func pt2)

real world:

KeyExchange.KEReal:1

(Pt1: WaitReq1, Pt2: WaitFwd1, Fw1: Init, Fw2: Init)

input guard: 4

control: environment

This shows the state of the session, which consists of the global context, the state of the world, the input guard and which entity has control. In the rest of this section, we will stop showing the entire global context and just show changes.

The state of the real world shows the state of each party, parameter and subfunctionality, including the values of state parameters. At this point, the parties, Pt1 and Pt2, and the forwarders, Fw1 and Fw2, are all in their initial states. An input guard of 4 means that all direct and adversarial messages sent to the real functionality will only be accepted if they are sent to ports with a port index less than 4, which is just an internal consistency check (port indices will be defined shortly). Finally, all sessions start with the environment in control.

While in a session, available actions depend on which entity has control:

* The environment can send a message to the functionality or transfer control to the adversary.
* The adversary can send a message to the functionality or transfer control to the environment.
* The functionality can execute a single step or run until it yields control to the environment or adversary.

Immediately following any of these actions, the script can assert something about its effect.

At any time in a session, the script can declare a variable, state an assumption or end the session with the finish command. Variables and assumptions inside a session are removed from the global context when the session ends. In Proof General, if you undo the finish command, it undoes the entire session.

Following a session, you can again declare a variable, state an assumption, exit the interpreter with the quit command or start another session. Reaching the end of the script file is the same as executing quit.

### Sending Messages (Port Indices)

The send command sends a message from a source port to a destination port and transfers control from the source entity (which must have control) to the destination entity. The general form is

src\_port@message\_path(args)@dest\_port

Since the environment has control in our sample script, the src\_port must be a port in the environment, namely pt1 or pt2 (or an expression that evaluates to one of them).

send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1)).

If the port is not the name of a variable, it must be enclosed in parentheses, as seen with the destination port, ((func, 1)). There are in fact two sets of parentheses, possibly because UC DSL requires one and the interpreter requires one.

A port, we can now exclusively reveal, consists of an addr and an int. The int is called a ***port index***. For environment ports, we generally don’t know or care what the port index is: we just declare variables like pt1 and pt2 with assumptions that they are in the environment. We also don’t know or care if pt1 and pt2 are equal. Port index 0 is used when transferring control between the environment and adversary and is implicitly used when a state machine executes a fail step and returns control to the environment with no message.

For a real functionality, the port index indicates the party, but not how you might think. Real and ideal functionalities implement a composite direct interface and a port index is assigned sequentially to each basic interface it contains, starting at 1. For example, in the KEDir composite interface, Pt1 has a port index of 1 and Pt2 has a port index of 2. Since the KEReal party Pt1 serves the interface KEDir.Pt1, direct messages to or from the Pt1 party use port ([1], 1), and those to or from the Pt2 party use port ([1], 2).

Ideal functionalities implement the same composite interfaces as real functionalities, even though they don’t have parties, so KEIdeal also uses ports ([1], 1) and ([1], 2). The port index numbering starts over from 1 within each composite interface, so in the Forwarding theory, the FwdDir.D interface is also assigned port index 1. The subfunctionalities KEReal.Fw1 and KEReal.Fw2 have distinct addresses and use ports ([1; 1], 1) and ([1; 2], 1), respectively.

For adversaries, including simulators, the interpreter assigns a unique port index to each instance (copy) of an adversarial interface in the real and ideal worlds. Adversarial port indices have to be globally unique because all simulators and adversaries used the same address, namely adv. We generally learn what indices are assigned by observing the output. For example, KeyExchange has three adversarial interfaces, two in the real world and one in the ideal. We’ll see below that KEI2S is assigned port index 1 and KEReal.Fw1.FwAdv and KEReal.Fw2.FwAdv are assigned 2 and 3, respectively.

When the real and ideal worlds are displayed outside of a session, the number shown after the name of each functionality is something like the lowest adversarial port index it uses, with those of subsimulators shown in parentheses. For KeyExchange, these are all 1. Within a session, the input guard is the smallest port index not assigned.

Message paths in interpreter scripts always consist of the names of the enclosing theory, composite interface (if applicable), component or basic interface and message type, separated by dots. This is different from message paths in the UC DSL, which omit the enclosing theory and sometimes substitute the name of a (sub-)functionality for that of an interface in order to make the destination port unnecessary (and hide its implementation). In KEReal, for example, the Pt1 party sends a fw\_req message to Fw1 with the path Fw1.D.fw\_req rather than Forwarding.FwDir.D.fw\_req. Fw1 implies the enclosing theory and the address of the destination (and D implies the port index). Similarly, KESim uses SMCReal.KE.KEI2S.ke\_sim\_req1 instead of KeyExchange.KEI2S. ke\_sim\_req1. In the interpreter, the source and destination ports are explicit for all messages, direct and adversarial, internal and external, and message paths are uniform.

The output of the send command is the effect of the command, the new session state and the message sent.

**send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1)).**

effect:

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

sending from environment

message: pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1))

In Proof General, the effect and file position are not shown in the \*configuration\* or \*response\* buffers, not least because they are always the same: a blank line and None.

One final note: for a functionality, the port index can be inferred from the message path, so an alternative syntax is to provide only the address and use $ instead of @. For example, the message above can also be written

**send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)$func.**

The environment transfers control to the adversary by sending it a message, but the message itself is unknown to the session (or the script writer). In this case, the message path is replaced with a single underscore and a port index of 0 is used for both ports:

**send (([], 0))@\_@((adv, 0)).**

To transfer control from the adversary to the environment, use

**send ((adv, 0))@\_@(([], 0)).**

### Executing the Functionality

When the real or ideal functionality or a simulator has control, it is executing one of its state machines. The run command tells the interpreter to execute statements until the functionality yields control to either the environment or the adversary. The output shows the effect of the execution, the current position in the UC DSL code and the session state.

**run.**

effect:

message was output: adversary:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 1), (func, 2), epdp\_port\_port\_key\_univ.`enc(pt1, pt2,

g ^ rand))

@((adv, 2))

;

UC file position: None;

state:

global context:

(…, rand : exp, Hrand : rand \in dexp)

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1,

Fw1: Wait ((func, 1), (func, 2), epdp\_port\_port\_key\_univ.`enc

(pt1, pt2, g ^ rand)), Fw2: Init]

input guard: 4

control: adversary

Here the effect was that the functionality sent a message to the adversary, specifically the fw\_obs message from the Fw1 functionality (whose address is func ++ [1]). In addition, the variable rand was added to the global context along with the assumption Hrand that rand was chosen from the dexp distribution, which is all we know about its value. The session state also shows that the Pt1 party is now in the WaitFwd2 state, the Fw1 subfunctionality is in the Wait state and Pt2 and Fw2 are still in their initial states. The UC file position is None because the functionality no longer has control.

The format of the output is meant to be easily parsed, e.g., by Proof General, which shows the effect in the \*response\* buffer and the rest of the output in the \*configuration\* buffer. The raw output above is shown in the \*uc dsl interpreter\* buffer.

The expected response at this point is for the adversary to approve sending the message:

**send ((adv, 2))@Forwarding.FwAdv.fw\_ok@((func ++ [1], 1)).**

global context: …

real world: …

input guard: 4

sending from adversary

message: ((adv, 2))@Forwarding.FwAdv.fw\_ok@((func ++ [1], 1))

**run.**

effect:

message was output: environment:

((func, 2))@KeyExchange.KEDir.Pt2.ke\_rsp1(pt1, g ^ rand ^ rand1)

@pt2

;

UC file position: None;

state:

global context:

(func : addr, …, rand : exp, Hrand : rand \in dexp, rand1 : exp,

Hrand1 : rand1 \in dexp)

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitReq2(pt2, rand1), Fw1: Final,

Fw2: Init]

input guard: 4

control: environment

The response from the Pt2 party completes the first part of the key exchange. The next four steps complete it:

**send** pt2@KeyExchange.KEDir.pt2.ke\_req2@((func, 2)).

global context: …

real world: …

input guard: 4

sending from adversary

message: pt2@KeyExchange.KEDir.pt2.ke\_req2@((func, 2))

**run.**

effect:

message was output: adversary:

((func ++ [2], 1))@Forwarding.FwAdv.fw\_obs((func, 2), (func, 1),

epdp\_key\_univ.`enc (g ^ rand1))@((adv, 3))

;

UC file position: None;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: Final, Fw1: Final,

Fw2: Wait((func, 2), (func, 1), epdp\_key\_univ.`enc (g ^ rand1))]

input guard: 4

control: adversary

**send ((adv, 3))@Forwarding.FwAdv.fw\_ok@((func ++ [2], 1)).**

global context: …

real world: …

input guard: 4

sending from adversary

message: ((adv, 3))@Forwarding.FwAdv.fw\_ok@((func ++ [2], 1))

**run.**

effect:

message was output: environment:

((func, 1))@KeyExchange.KEDir.Pt1.ke\_rsp2(g ^ rand ^ rand1)@pt1

;

UC file position: None;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: Final, Pt2: Final, Fw1: Final, Fw2: Final]

input guard: 4

control: environment

Compare this output to Figure 3. The interpreter output is missing messages exchanged internally to the functionality. These can be seen by using the step command instead of run. Starting from the first send command in the script, it takes eight steps to reach the first output message to the adversary:

**step. (\* deliver the message \*)**

effect:

;

UC file position: …/KeyExchange.uc 2285 2939;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitReq1, Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

running at []: KeyExchange.KEReal/Pt1/WaitReq1

local context:

(envport -> envport func, q1 -> witness, intport Pt1 -> (func, 1), intport Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

;

This step delivers the message to the destination port. There is no effect yet, but the UC file position reflects the statement about to be executed. (In Proof General, the byte offset range 2285-2939 is converted to a highlighted region of the KeyExchange.ec file shown in uc-file-buffer in its own window (lines 65-77).) The session state has not changed yet, either, except now there is a ***local context*** showing

* the current definition of the UC DSL version of envport as a one-parameter function relative to func
* local variables declared in the state (q1, initialized to witness)
* parameters of the current state of the state machine, which for a real functionality include the internal ports of the functionality’s parties
* pattern variables of the active match message statement branch (pt1 and pt2)

**step. (\* if (envport pt2) { … } \*)**

effect:

;

UC file position: …/KeyExchange.uc 2316 2327;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitReq1, Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

running at []: KeyExchange.KEReal/Pt1/WaitReq1

local context:

(envport -> envport func, q1 -> witness, intport Pt1 -> (func, 1), intport Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

;

**step. (\* q1 <$ dexp; \*)**

effect:

note: random value was assigned to: rand

;

UC file position: …/KeyExchange.uc 2713 2903;

state:

global context: (…, rand : exp, Hrand : rand \in dexp)

real world:

KeyExchange.KEReal:1

[Pt1: WaitReq1, Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

running at []: KeyExchange.KEReal/Pt1/WaitReq1

local context:

(envport -> envport func, q1 -> rand, intport Pt1 -> (func, 1), intport Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

;

Here the effect section “notes” that a random value was generated as the global variable rand, which (with Hrand) was added to the global context and assigned as the value of q1 in the local context.

**step. (\* send … and transition …. \*)**

effect:

;

UC file position: None;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

sending from []: KeyExchange.KEReal

message:

((func, 1))@

Forwarding.FwDir.D.fw\_req

((func, 2), epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand))

@((func ++ [1], 1))

;

This step sends a message to subfunctionality Fw1, which has not yet been delivered. The UC file position is None because the interpreter is routing a message, not executing code. The Pt1 state is now WaitFwd2 but it is no longer executing. The Fw1 subfunctionality is still in its Init state and is not yet executing. There is no local context.

**step. (\* deliver the message \*)**

effect:

;

UC file position: …/Forwarding.uc 2071 2948;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

running at [1]: Forwarding.Forw/Init

local context:

(envport -> envport (func ++ [1]), pt1 -> (func, 1), pt2 -> func, 2), u -> epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand))

;

This step delivers the message to subfunctionality Fw1, which is reflected in both the UC file position and the “running at” line. The local context is for the Init state of Fw1. Note the altered definition of envport: from here, func is considered part of the environment because it is not inside func ++ [1] or adv.

**step. (\* if (envport pt2) { … } \*)**

effect:

;

UC file position: …/Forwarding.uc 2781 2887;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1, Fw1: Init, Fw2: Init]

input guard: 4

running at [1]: Forwarding.Forw/Init

local context:

(envport -> envport (func ++ [1]), pt1 -> (func, 1), pt2 -> func, 2), u -> epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand))

;

**step. (\* send … and transition … \*)**

effect:

;

UC file position: None;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1,

Fw1:

Wait

((func, 1),(func, 2),

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand)),

Fw2: Init]

input guard: 4

sending from [1]: Forwarding.Forw

message:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 2), (func, 1),

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand))

@((adv, 2))

;

**step. (\* deliver the message \*)**

effect:

message was output: adversary:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 1), (func, 2),

epdp\_port\_port\_key\_univ.`enc(pt1, pt2, g ^ rand))

@((adv, 2))

;

UC file position: None;

state:

global context:

(func : addr, …, rand : exp, Hrand : rand \in dexp)

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitFwd1,

Fw1:

Wait

((func, 1), (func, 2),

epdp\_port\_port\_key\_univ.`enc(pt1, pt2, g ^ rand)),

Fw2: Init]

input guard: 4

control: adversary

The output of the last step command is the same as the output of the run command used previously; the adversary has control and there is no local context.

When the script sends a null message to transfer control between the adversary and the environment, a step or run command is still needed:

**send ((adv, 0))@\_@(([], 0)).**

effect:

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

sending from adversary

**run.**

effect:

message was output: environment: ((adv, 0)@\_@(([], 0))

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

control: environment

This is the only time step or run can be used when the functionality is not in control.

### Asserting Effects

An ***assertion*** lets you verify that a send, run or step command had the expected effect. It has one of these forms:

assert rand.

assert fail\_out.

assert msg\_out sent-message entity-with-control.

assert ok.

The first form is true after a step command executes a random assignment statement. The second is true after a step or run command executes a fail statement or tries to deliver a message that is not allowed, such as an adversarial message to a functionality that is in its initial state, an adversarial message sent by the environment or a direct message sent by the adversary. This effect also indicates that the environment now has control. The third form is true after a step or run command delivers a message to the environment or adversary, if the message exactly matches the one specified in the assertion and the entity with control is correct. The entity with control is specified as either ctrl\_env or ctrl\_adv. The last form is true whenever none of the other conditions apply: after (1) a send command; or (2) a step command that (a) delivers a message to the functionality; (b) executes anything other than a random assignment (<$) or fail statement, namely a send-transition or a regular assignment (<-) statement; or (c) evaluates the condition of an if, elif or match statement.

If an assertion is true, it produces no output. If it is false, it generates an error that must be corrected for the script to run. For example, the last step of the real session in Figure 12 is

assert msg\_out

((func, 1))@KeyExchange.KEDir.Pt1.ke\_rsp2(g ^ rand1 ^ rand)@pt1

ctrl\_env.

which is true. An assertion is not counted as a command, literally: the numeric part of the command prompt is not incremented.

If we continue the real session by having the environment try to initiate a second key exchange, the message is delivered to the Pt1 party, which rejects it because it is in its Final state:

**send pt2@KeyExchange.KEDir.Pt1.ke\_req1(pt1)@((func, 1)).**

effect:

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

sending from environment

message: pt2@KeyExchange.KEDir.Pt1.ke\_req1(pt1)@((func, 1))

**run.**

effect:

note: "fail." was called.

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

control: environment

We can confirm this (or that we expected this) with an assertion:

**assert fail\_out.**

Again, there is no output from the assertion because it is correct.

## Ideal World Sessions

In Figure 12, we next start an ideal session. The output is very similar to starting a real session.

**ideal.**

global context:

(func : addr, adv : addr, IncFuncAdv : inc func adv, pt1 : port,

pt2 : port, pt1\_env : envport func pt1, pt2\_env : envport

func pt2)

Ideal world:

KeyExchange.KEIdeal:1[WaitReq] /

KeyExchange.KESim:1[uninitialized/WaitReq1]

input guard: 4

control: environment

The state of the ideal world shows the state of the ideal functionality and the state of the simulator, including any subsimulators it contains. Until the simulator receives its first message from the ideal functionality, its state is annotated as uninitialized because it does not know the address of the ideal functionality. It is an error if it receives a message from the adversary while uninitialized.

The example ideal session sends the same messages to the functionality as the real session, starting with

**send pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1)).**

effect:

;

UC file position: None;

state:

global context: …

ideal world: …

input guard: 4

sending from environment

message: pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1))

**run.**

effect:

message was output: adversary:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 1), (func, 2), epdp\_port\_port\_key\_univ.`enc(pt1, pt2,

g ^ rand))

@((adv, 2))

;

UC file position: None;

state:

global context:

(func : addr, …, rand : exp, Hrand : rand \in dexp)

ideal world:

KeyExchange.KEIdeal:1[WaitSim1(pt1, pt2)] /

KeyExchange.KESim:1[initialized: func/WaitAdv1(rand)]

input guard: 4

control: adversary

The effect is the same as in the real session, as is the global context. In the ideal world, KEIdeal is now in state WaitSim1 and KESim is initialized with the address func and is in state WaitAdv1. If instead we step through the execution, the internal details differ:

**step. (\* deliver the message \*)**

effect:

;

UC file position: …/KeyExchange.uc 6677 7313;

state:

global context: …

ideal world:

KeyExchange.KEIdeal:1[WaitReq1] /

KeyExchange.KESim:1[uninitialized/WaitReq1]

input guard: 4

running at 1: KeyExchange.KEIdeal:1/WaitReq1

local context:

(envport -> envport func, pt1 -> pt1, pt2 -> pt2)

The message was delivered and a local context set up. The next two steps execute code that verifies that the port specified in the message is in the environment and sends a message to the simulator.

**step. (\* if (envport pt2) { … } \*)**

effect:

;

UC file position: …/KeyExchange.uc 7213 7281;

state:

global context: …

ideal world: …

input guard: 4

running at 1: KeyExchange.KEIdeal:1/WaitReq1

local context: …

**step. (\* send … and transition …. \*)**

effect:

;

UC file position: None;

state:

global context: …

ideal world:

KeyExchange.KEIdeal:1[WaitSim1(pt1, pt2)] /

KeyExchange.KESim:1[uninitialized/WaitReq1]

input guard: 4

sending from KeyExchange.KEIdeal

message:

((func, 1))@KeyExchange.KEI2S.ke\_sim\_req1(pt1, pt2)@((adv, 1))

The ideal functionality has transitioned to the WaitSim1 state and there is no local context as the interpreter routes the message. The next step delivers it:

**step. (\* deliver the message \*)**

effect:

;

UC file position: …/KeyExchange.uc 9485 9496;

state:

global context: …

ideal world:

KeyExchange.KEIdeal:1[WaitSim1(pt1, pt2)] /

KeyExchange.KESim:1[initialized: func/WaitReq1]

input guard: 4

sending from KeyExchange.KEIdeal

local context:

(envport -> envport func, q1 -> witness, intport KEReal.Pt1 -> (func, 1),

intport KEReal.Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

Delivering the message initializes the simulator with the address of the ideal functionality, func, and builds a local context for the simulator. In the next step, the simulator generates a random value and assigns it to the q1 variable:

**step. (\* q1 <$ dexp; \*)**

effect:

note: random value was assigned to: rand

;

UC file position: …/KeyExchange.uc 9805 9988;

state:

global context:

(func : addr, …, rand : exp, Hrand : rand \in dexp)

ideal world:

KeyExchange.KEIdeal:1[WaitSim1(pt1, pt2)] /

KeyExchange.KESim:1[initialized: func/WaitReq1]

input guard: 4

running at 1: KeyExchange.KESim/WaitReq1

local context:

(q1 -> rand, intport KEReal.Pt1 -> (func, 1),

intport KEReal.Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

The random value has been added to the global context as rand and assigned as the value of q1 in the local context. The next two steps send a message to the “real” adversary and deliver it.

**step. (\* send … and transition … \*)**

effect:

;

UC file position: None;

state:

global context: …

ideal world:

KeyExchange.KEIdeal:1[WaitSim1(pt1, pt2)] /

KeyExchange.KESim:1[initialized: func/WaitAdv1(rand)]

input guard: 4

sending from 1: KeyExchange.KESim

message:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 1), (func, 2),

epdp\_port\_port\_key\_univ.`enc (pt1, pt2, g ^ rand))

@((adv, 2))

;

**step. (\* deliver the message \*)**

effect:

message was output: adversary:

((func ++ [1], 1))@

Forwarding.FwAdv.fw\_obs

((func, 1), (func, 2),

epdp\_port\_port\_key\_univ.`enc(pt1, pt2, g ^ rand))

@((adv, 2))

;

UC file position: None;

state:

global context: …

ideal world: …

input guard: 4

control: adversary

The output of this step is the same as the output of the run step; the adversary has control and there is no local context. The ideal session executed seven steps before yielding control, whereas the real session executed eight. The internal messaging between the ideal functionality and simulator differed from the real internal messaging, but as far as the environment can tell, they were identical.

When the last session is finished, the script can end with an explicit quit command or just end. The quit command shuts down the interpreter and any subsequent commands are ignored.

## Resolving Blocks

The UC DSL interpreter uses the simplify, rewrite and smt tactics of EasyCrypt automatically to execute the statements of a state machine symbolically. Sometimes it encounters expressions that it cannot reduce sufficiently to determine what to do next, such as a source or destination port of a message, the distribution of a random selection or the condition of an if, elif or match statement. This is called ***blocking***. For example, in a key exchange between pt1 and pt2 initiated by pt1, it is required that pt2 be the source port of the message that kicks off the second half of the exchange. This is enforced by these lines in KEReal, Party Pt2, state WaitReq2:

if (pt2’ = pt2) { … }

else { fail. }

where pt2 is from the original ke\_req1 message and pt2’ sent the ke\_req2 message. The script in Figure 12 complies, but what if instead pt1 sends it?

**send pt1@KeyExchange.KEDir.Pt2.ke\_req2@((func, 2)).**

effect:

;

UC file position: None;

state:

global context: …

real world: …

input guard: 4

sending from environment

message: pt1@KeyExchange.KEDir.Pt1.ke\_req1(pt2)@((func, 1))

**step.**

effect:

;

UC file position: …/KeyExchange.uc 5308 5468;

state:

global context: …

real world:

KeyExchange.KEReal:1

[Pt1: WaitFwd2(pt1, rand), Pt2: WaitReq2(pt2, rand1), Fw1: Final,

Fw2: Init]

input guard: 4

running at KeyExchange.KEReal/Pt2/WaitReq2

local context:

(q1 -> witness, intport KEReal.Pt1 -> (func, 1),

intport KEReal.Pt2 -> (func, 2), pt1 -> pt1, pt2 -> pt2)

**step.**

[error:0-4]

blocking: cannot decide if condition

The first step delivers the message, while the second displays only the error message. The state of the functionality is not changed, so it is not repeated. The state tells us that the block occurred when trying to evaluate character positions 5308-5468 in the KeyExchange.uc file, which is precisely the conditional statement above. The interpreter has a debug mode that will show what the interpreter was trying to do.

**debug.**

[warning:0-0]

debugging messages turned on (see buffer \*trace\*)

**step.**

<dbg>

trying to prove truth or falsity of

pt1 = pt2

</dbg>

<dbg>

formula simplified to

pt1 = pt2

</dbg>

<dbg>

unable to prove formula or its negation

</dbg>

[error:0-4]

blocking: cannot decide if condition

(Proof General shows the debug output in the \*response\* buffer but then replaces it with the error message. Switch to the \*trace\* or \*uc dsl interpreter\* buffer to see it again.) As expected, it is trying to prove or disprove pt1 = pt2 but can’t because it has no information either way. Humans might assume that the variables pt1 and pt2 have different values, but EasyUC doesn’t.

We can resolve the block by adding an assumption. We can add it at the top of the script or just before the error happens (making it local to this session):

**assumption pt1\_is\_not\_pt2 : pt1 <> pt2.**

UC file position: …

state:

global context: (…, pt1\_is\_not\_pt2 : pt1 <> pt2)

…

**step.**

<dbg>

trying to prove truth or falsity of

pt1 = pt2

</dbg>

<dbg>

formula simplified to

false

</dbg>

<dbg>

formula’s negation proved

</dbg>

effect:

;

UC file position: …/KeyExchange.uc 5461 5467;

state:

global context: …

**debug.**

[warning:0-0]

debugging messages turned off

The block has been resolved and the state has been updated (the new file position is the fail statement in the else clause). Debug mode is turned off by repeating the command (i.e., it’s a toggle).

Lemmas, user reductions (simplify hints) and rewrite hints can be added to the model to prevent or remove blocks. This is done in an EasyCrypt support theory that the EasyUC model ec\_requires. Rewrite hints should be added to the UCBasicTypes.ucdsl\_interpreter\_hints rewrite database. (Note: to run the support theory on its own, you will need to tell EasyCrypt where to find the UCBasicTypes theory, such as with the command-line option -I <path to EasyUC>/uc-dsl/prelude or via Emacs and Proof General variables.)

Global SMT settings can be set using the same syntax as in EasyCrypt:

prover smt-info.

where smt-info is one or more of

* [prover-selector] or prover=[prover-selector]: a list of provers to run, as a list of strings in square brackets and separated by spaces
  + Recall easycrypt config lists known provers, among other things
  + Prover names can omit the version or specify all or part of it: "Alt-Ergo", "Alt-Ergo2" and "Alt-Ergo2.4.2" are all accepted, as of this writing
  + To modify the global selection (see below), a string can be prefixed with + to add the prover to the list or - to remove it
* quorum=n: the minimum number of provers that must return success for the tactic to succeed
* timeout=n: the maximum time (in integer seconds) each prover has to respond
* maxprovers=n: the maximum number of provers to run in parallel
* verbose: set the verbosity level; also verbose=n, but n seems to have no effect
  + The tactic reports how long it ran
* selected: list the EasyCrypt axioms and lemmas that were sent to the SMT prover
* wantedlemmas=dbhint: select only the specified axioms and lemmas
  + A list of axioms and lemmas separated by spaces
  + An axiom or lemma can be excluded by prefixing it with – (or use unwantedlemmas)
  + All axioms and lemmas of theory T can be included (or excluded) using @T (or -@T)
* unwantedlemmas=dbhint: exclude the specified axioms, lemmas and theories
* all: select all available lemmas except unwantedlemmas and those marked nosmt
  + This option causes the wantedlemmas option to be ignored
* n or maxlemmas=n: the maximum number of lemmas that may be selected
  + This option may be broken in the current version
* iterate: incrementally augment the number of selected lemmas
* lazy: how this affects the smt tactic and/or SMT provers is unclear
* full: how this affects the smt tactic and/or SMT provers is unclear

Note that the EasyCrypt options debug and dump in are not available. In addition, verbose and selected appear to be ignored.

Tips

* An interpreter script should provide specific, literal values wherever possible in the messages it sends to the functionality. If a message has a list parameter, for example, specify [] or [1; 2; 3] , or define a variable s and give it a value with an assumption:

var s : int list.

assumption s\_def : s = [].

You can assume properties of a variable, such as

assumption s\_sz : size s = 5.

but assigning a literal value is generally better. Scripts are basically test cases.

* To give the interpreter access to an operator’s definition, simply add axiomatized by xxx\_def to the definition and make the axiom a user reduction (hint simplify xxx\_def). This will effectively unfold the operator whenever it appears and is particularly applicable for constants (nullary operators).
* Inductive types and operators defined inductively over their constructors work very well, because simplify (specifically iota) can reduce them. For example, EasyCrypt reduces size 1 :: 2 :: 3 :: [] (or its abbreviation, size [1; 2; 3]) to 3 because size is defined inductively over [] and ::. If an operator is defined in some other way, one should then prove lemmas for applying it to each constructor of one of its arguments and make them user reductions. If you can, define new types and operators inductively.
* Lemmas over the “effective” constructors of a type are just as useful as over true constructors. These can often be identified by looking for (or proving your own) induction axioms or lemmas (e.g., those ending in “ind” or “W”).
  + In the SmtMap theory, the operators empty and "\_.[\_<-\_]" (point assignment) can construct any element of type fmap (so says the induction axiom fmapW). They aren’t the only way, but are arguably the most natural way, and an expression like empty.[1<-15].[2<-12] is a reasonable choice for a “literal” fmap.
  + In the FSet theory, the lemma fset\_ind tells us that the operators fset0 and fun s x => s `|` (fset1 x) (add one element) are effective constructors of type fset. Analogously, in the Logic theory, unary predicates of type 'a -> bool considered as characteristic functions of finite sets have pred0 and predU1 as effective constructors.
  + The effective constructors of the natural numbers are 0 and S (successor, meaning fun n => n + 1), according to the axiom intind. For example, in the Ring theory one finds a definition for the exponentiation operator, exp, and the lemmas

lemma expr0 x : exp x 0 = oner.

lemma exprS x i :

0 <= i => expr x (i+1) = x \* (exp x i).

lemma expr\_pred x i :

0 < i => expr x i = x \* (exp x (i-1)).

In practice, expr\_pred is more useful because it can be matched to an expression such as exp 3 5 automatically, whereas exprS requires a proof term, (exprS \_ 4), which an interpreter script has no way to provide. Both lemmas reduce it to 3 \* (exp 3 4) and repeated applications eventually lead to 243, because simplify (specifically logic) can multiply literal ints and do other simple math.

* + In the List theory, drop is defined inductively over its list argument, but it is equally useful to have lemmas over its int argument: drop0 and dropS (or dropP). Inconveniently, dropS and dropP are not in List, so you will need to add them to your own support theory.
* Operators such as witness and choiceb cannot be reduced to literal values because they are necessarily underspecified. This becomes a problem when that’s what you need to continue computing. For example, the fdom operator returns the domain of an fmap as an fset, but its definition is oflist (to\_seq (dom m)) and the definition of to\_seq uses choiceb. Even though the domain of a given fmap is unique, EasyCrypt cannot tell you what it is as, say, an expression of the form fset0 `|` (fset1 1) `|` (fset1 2).
* User reductions are the main way to help the interpreter execute a functionality, but not all lemmas can be used that way. In particular, a lemma with a precondition will only be applied if the precondition can be proved without using other user reductions. A precondition instantiated as 0 <= 5, for example, reduces to true because simplify (specifically logic) can do it, but 0 <= exp 2 5 does not reduce. A lemma that uses exp in a precondition should be added to the ucdsl\_interpreter\_hints rewrite database, because the rewrite tactic will simply add 0 <= exp 2 5 as a subgoal and get to it later.
* Keep hints in their own file so they don’t interfere with proof lemmas. That is, if you define hints as you go, subsequent uses of the simplify tactic will automatically use any applicable user reductions. If you later change the hints, you may have to change the proof to add steps that used to be user reductions or delete steps that have become user reductions. To keep the proof environment stable, keep hints separate from proofs and don’t require hint files.
* Don’t name the hints file Hints.ec. If you organize your models and theories into multiple directories and each one has a Hints.ec file, only one of them will be loaded (the first one found when the script reaches the load command and encounters the first require Hints directive). Likewise, if the Hints.ec file does require import on a lot of other theories that have lemmas with the same name, make sure your hints use qualified names to specify the right lemma (or don’t use import and use qualified names for everything).
* When the interpreter blocks, copy the expression it can’t reduce into a file and make it a lemma (if it’s not of type bool, add = witness or whatever you think the correct reduced expression is). Require whatever theories you need and now you can work on the block in EasyCrypt instead of restarting the interpreter and rerunning the script every time you change something. Add hints and lemmas to the new file until the lemma can be proved with just simplify, rewrite ucdsl\_interpreter\_hints and smt steps. Finally, move the lemmas to the support theory to which they logically belong, hints to the hint file and smt directives to the interpreter script.
* By stepping through your model in the interpreter script, you may be able to notice expressions not reducing before they cause a block. For example, assignment statements store the reduction of the right-hand side in the variable(s) of the left-hand side. If the local context doesn’t show values reduced as far as you think they should be, fix it now because the problem will only get worse if the unreduced expression is used in later statements.

# UC DSL Quick Reference

This section offers a quick reference to the UC DSL with a minimum of explanation and examples. It also covers the UC DSL interpreter syntax.

## Lexical Notes

Comments are enclosed in (\* and \*). They nest, as they do in EasyCrypt.

UC DSL identifiers consist of letters, numbers, apostrophes and underscores. They must start with a letter and may not end in an underscore or contain consecutive underscores. They must also not start with "uc\_" or "UC\_". The thing being named imposes additional restrictions as follows:

* Things whose names must start with an uppercase letter:
  + Files (a.k.a. theories)
  + Interfaces, including the subinterfaces of composite interfaces, such as D in FwDir
  + Functionalities, including subfunctionalities, such as Fw1 in KEReal
  + Parties
  + States
* Things whose names must start with a lowercase letter.
  + Message types
  + Ports
  + Parameters of message types and states
  + Local variables in states

The following UC DSL keywords are reserved (cannot be used as identifiers). Note that they are exclusively lowercase.

* adversarial, and, direct, ec\_requires, elif, else, end, envport, fail, functionality, if, implements, in, initial, intport, match, message, out, party, send, serves, simulates, simulator, state, subfun, transition, uc\_requires, uses, var, with

The following keywords from EasyCrypt are also reserved.

* Pr, Top, abbrev, abstract, admit, algebra, alias, apply, as, assert, assumption, auto, axiom, axiomatized, beta, by, byequiv, byphoare, bypr, call, case, cbv, cfold, change, class, clear, clone, congr, conseq, const, cut, debug, declare, delta, do, done, eager, elim, equiv, eta, exact, exfalso, exists, export, fel, fission, for, forall, fun, fusion, glob, goal, have, hint, hoare, idtac, import, include, inductive, inline, instance, iota, is, islossless, kill, lemma, let, local, logic, modpath, module, move, nosmt, notation, of, op, phoare, pose, pr, pragma, pred, print, proc, progress, proof, prover, qed, rcondf, rcondt, realize, reflexivity, remove, rename, replace, require, res, return, rewrite, rnd, rwnormal, search, section, seq, sim, simplify, skip, smt, sp, split, splitwhile, subst, suff, swap, symmetry, then, theory, time, timeout, transitivity, trivial, try, type, undo, unroll, while, why3, wp, zeta

The UC DSL interpreter adds no reserved words—all of its keywords are either already reserved by EasyUC and/or EasyCrypt or are not reserved (may still be used as identifiers).

## UC DSL Syntax

This section presents the UC DSL syntax semi-formally. The only meta-syntax used is square brackets ([]) to indicate optional elements. Plural nouns indicate that a list of elements is permitted.

1. Preamble

uc\_requires UC-DSL-theory-names.

where UC-DSL-theory-name is the name of a ".uc" file on the load path (but omitting ".uc"). This also requires all UC DSL and EasyCrypt files required (directly or indirectly) by the named files. Elements of these theories are referenced using qualified names.

ec\_requires [+]EasyCrypt-theory-names.

where EasyCrypt-theory-name is the name of an ".ec" file on the load path (but omitting ".ec"). This also requires all EasyCrypt files required (directly or indirectly) by the named files. The optional "+" means to import the theory as well, allowing the use of unqualified names (but does not import anything it imports).

1. Interface
   1. Basic interface

direct ID { message-types }

adversarial ID { message-types }

where ID is the name of the interface and must start with an uppercase letter.

* 1. Message type

in [ID2@] ID1 [([message-parameters])]

out ID1 [([message-parameters])] [@ID2]

where ID1 is the name of the message type, ID2 is the name of a port and both must begin with a lowercase letter. Message types in direct interfaces must specify a port and message types in adversarial interfaces must not specify a port. The in and out keywords indicate the direction of the message, relative to a real or ideal functionality that implements them.

* 1. Message parameter

ID : TYPE

where ID is the name of the parameter, which must begin with a lowercase letter, and TYPE is an EasyCrypt type expression. Multiple parameters are separated by commas.

* 1. Composite interface

direct ID { subinterfaces }

adversarial ID { subinterfaces }

where ID is the name of the interface and must start with an uppercase letter.

* 1. Subinterface

ID1 : ID2

where ID1 is the name of the subinterface, which must start with an uppercase letter, and ID2 is the name of a basic interface. A direct or adversarial composite interface must only contain direct or adversarial basic subinterfaces, respectively.

1. Ideal functionality

functionality ID1 implements ID2 ID3 { state-machine }

where ID1 is the name of the functionality, which must begin with an uppercase letter, ID2 is the name of a composite direct interface and ID3 is the name of a basic adversarial interface.

1. Real functionality

functionality ID1 [([parameters])] implements ID2 [ID3] {  
 [subfunctionalities] parties }

where ID1 is the name of the functionality, which must begin with an uppercase letter, ID2 is the name of a composite direct interface and ID3 is the name of an optional composite adversarial interface.

* 1. Parameter

ID1 : ID2

where ID1 is the name of the parameter, which must start with an uppercase letter, and ID2 is the name of a composite direct interface. Multiple parameters are separated by commas.

* 1. Subfunctionality

subfun ID1 = ID2

where ID1 is the name, which must start with an upper case letter, of a new instance (clone) of the ideal functionality named ID2 (which must be a qualified name).

* 1. Party

party ID1 serves [ID2] [ID3] { state-machine }

where ID1 is the name of the party, which must begin with an uppercase letter, ID2 is the name of an optional basic direct interface and ID3 is the name of an optional basic adversarial interface. Each subinterface of the composite interface implemented by the functionality must be served by exactly one party. Note that the serves keyword is required even if ID2 and ID3 are both omitted.

1. Simulator

simulator ID1 uses ID2 simulates ID3 { state-machine }

where ID1 is the name of the simulator, which must start with an uppercase letter, ID2 is the name of the basic adversarial interface of an ideal functionality and ID3 is the name of a real functionality (with names of ideal functionalities as arguments for any parameters it has) whose adversarial communications ID1 will simulate.

1. State Machine
   1. A state machine is a non-empty list of states.

[initial] state ID [([parameters])] {

[local-variables] state-body }

where ID is the name of the state, which must begin with an uppercase letter.

* + 1. There must be exactly one initial state.
    2. The initial state cannot have parameters.
    3. A state parameter has a name and a type. Parameter names must start with a lowercase letter. Types are EasyCrypt type expressions. Multiple parameters are separated by commas.
  1. Local variable

var IDs : TYPE;

where IDs is a list of the names of the variables, separated by commas, and TYPE is an EasyCrypt type expression. Variable names must start with a lowercase letter.

* 1. State body

match message with

[|] [ID@]msg-path[(parameter-args)] => { statements }

| \* => { fail. }

end

where

* + 1. ID is a *pattern variable* that is set to the source port from which the message came (for direct messages from the environment only; i.e., excluding both adversarial messages and direct messages from subfunctionalities or parameters). It is an identifier that must start with a lowercase letter.
    2. msg-path is a *message path*, which is a dot-delimited sequence of identifiers starting with the name of a functionality parameter, subfunctionality, composite interface or basic interface and ending with the name of a message type.
    3. “\*” is a wildcard pattern matching any message. It can also occur after a partial message path to match all possible completions (in which case pattern variables for the port and arguments must be omitted). The body of a wildcard can only be unconditional failure.
    4. Vertical bars separate message match clauses and one may optionally occur before the first one.
    5. parameter-args is a comma-separated list of identifiers called *pattern variables*, which must all start with a lowercase letter. Each is set to the value of the corresponding typed expression provided when the message was sent. A parameter-arg may also be an underscore, in which case the corresponding expression is discarded.
    6. All possible in message paths defined by the interfaces served, implemented or used by the state machine, and direct out paths from subfunctionalities in the case of parties, must be matched by at least one pattern in the match statement, except that the initial state cannot receive adversarial messages.
    7. Each pattern must be reachable, meaning at least one message that matches the pattern does not match any previous pattern (e.g., no duplicate patterns).
  1. Statement
     1. Assignment of a value (i.e., a typed expression) to a variable, tuple or map key

x <- 0;

(x, y) <- (y, x);

m.[x] <- y;

* + 1. Assignment of a value sampled from a (sub-)distribution to a variable, tuple or map key

x <$ [5..10];

* + 1. Match statement

match exp with

[|] pattern => { statements }

[ | … ]

end

where exp is an EasyCrypt typed expression whose value belongs to an inductive type and pattern is an EasyCrypt pattern consisting of a constructor of the type applied to pattern variables. These variables may be used in the corresponding statement block. There must be one pattern-statement pair for each constructor of the type.

* + 1. Conditional statement

if (b) { statements }

[ elif (b2) { statements } ]

[ else { statements } ]

where b and b2 are EasyCrypt typed expressions of type bool. Braces are required even if a branch consists of a single statement. Any number of elif branches may be used.

* + 1. Transfer of control (send a message and transition to a new state)

send M [([exps])] [@[(]destport[)]]

and transition S [([exps])].

where M is a message path, S is a non-initial state of the functionality, destport (for direct messages only) is an EasyCrypt typed expression of type port and exps is a comma-separated list of EasyCrypt typed expressions providing arguments for the message and/or state parameters. If destport is an identifier, parentheses are optional. In an ideal functionality, the initial state can only send adversarial messages.

* + 1. Failure (return control to the "root" of the environment without a message and without changing the current state)

fail.

## EPDPs

An encoder and partial decoder pair (EPDP) is a generic mechanism for converting values of one type into values of another. One or more EPDPs are needed whenever a UC DSL model makes explicit use of the univ type, as the functionalities above do when using the Forw functionality. This section provides additional information on EPDPs and pointers to the relevant EasyCrypt code.

Every UC DSL file implicitly requires and imports the EasyCrypt files in the uc-dsl/prelude folder where EasyUC is installed. These include

* UCBasicTypes – exports UCEncoding and UCUniv
* UCEncoding – defines EPDPs in general. Basic definitions include

(\* a record type definition for EPDPs \*)

type ('a, 'b) epdp = {enc: 'a -> 'b; dec : 'b -> 'a option}.

(\* a validity condition all EPDPs should satisfy \*)

op valid\_epdp (epdp : ('a, 'b) epdp) : bool =

(forall (x : 'a), epdp.`dec (epdp.`enc x) = Some x) /\

(forall (y : 'b, x : 'a),

epdp.`dec y = Some x => epdp.`enc x = y).

(\* the identity EPDP \*)

op epdp\_id : ('a, 'a) epdp =

{|enc = fun x => x; dec = fun y => Some y|}.

(\* composition of two EPDPs \*)

op epdp\_comp (epdp2 : ('b, 'c) epdp,

epdp1 : ('a, 'b) epdp) : ('a, 'c) epdp =

{|enc = (\o) epdp2.`enc epdp1.`enc;

dec = fun z =>

match epdp2.`dec z with

| None => None

| Some y => epdp1.`dec y

end|}.

with lemmas that prove epdp\_id is valid and the result of composing two valid EPDPs is valid. It also defines generic operators for constructing EPDPs for enumerated types of size 2 to 8, tuple types of size 2 to 8, option types and list types from EPDPs for their components (e.g., given an ('a, 'b) epdp and a ('c, 'd) epdp, construct an ('a \* 'c, 'b \* 'd) epdp, an ('a list, 'b list) epdp and so on).

* UCUniv – defines the univ type,

type univ = bool list.

and constant EPDPs for encoding and decoding unit, bool, int, tuples of univ values of size 2 to 8, enumerations of univ values of size 2 to 8, univ option and univ list to and from univ . It also defines generic operators that construct EPDPs to and from univ for enumerated types of size 2 to 8, tuple types of size 2 to 8, option types and list types from EPDPs for their components (e.g., given an ('a, univ) epdp and a ('b, univ) epdp, construct an ('a \* 'b, univ \* univ) epdp, an ('a list, univ list) epdp and so on).

Note that the univ representations of all these types are not disjoint: some univ values can be decoded in multiple ways, so it is required to know a priori what EPDP was used. For example, KEReal.Pt1 and KEReal.Pt2 both use epdp\_port\_port\_key\_univ to exchange a message containing two ports and a key via the Fw1 subfunctionality but use epdp\_key\_univ for the message containing just a key sent via Fw2.

The EPDPs used by KEReal , KESim, SMCReal and SMCSim are defined in the KeysExponentsAndPlaintexts.ec file in the examples/smc-case-study folder. In that file, the text and exp types are abstract, so their EPDPs are too, requiring axioms stating that they are valid:

type exp.

op epdp\_exp\_univ : (exp, univ) epdp.

axiom valid\_epdp\_exp\_univ : valid\_epdp epdp\_exp\_univ.

type text.

op epdp\_text\_key : (text, key) epdp.

axiom valid\_epdp\_text\_key : valid\_epdp epdp\_text\_key.

If in some other context the theory were cloned and definitions provided, those axioms would become lemmas (proof obligations of the cloning operation).

The key type is also abstract, but there is a bijection from exp to key named gen (the inverse is log):

type key.

op gen (q : exp) : key = g ^ q.

op gen\_rel (x : key) (q : exp) : bool = x = g ^ q.

op log (x : key) : exp = choiceb (gen\_rel x) e.

lemma gen\_log : cancel gen log. (\* proof omitted \*)

lemma log\_gen : cancel log gen. (\* proof omitted \*)

(where cancel f g means g(f(x)) = x.) An EPDP between exp and key is constructed using an operator in UCEncoding not mentioned above,

op epdp\_bijection (f : 'a -> 'b, g : 'b -> 'a) : ('a, 'b) epdp =

{|enc = f; dec = (\o) Some g|}.

lemma valid\_epdp\_bijection (f : 'a -> 'b, g : 'b -> 'a) :

cancel f g => cancel g f => valid\_epdp (epdp\_bijection f g).

and then composed with epdp\_exp\_univ to give an EPDP between key and univ:

op epdp\_key\_univ : (key, univ) epdp =

epdp\_comp epdp\_exp\_univ (epdp\_bijection log gen).

An EPDP between text and univ is also defined by composition, and the remaining EPDP uses a tuple constructor from UCUniv:

op epdp\_text\_univ : (text, univ) epdp =

epdp\_comp epdp\_key\_univ epdp\_text\_key.

op epdp\_port\_port\_key\_univ : (port \* port \* key, univ) epdp =

epdp\_tuple3\_univ epdp\_port\_univ epdp\_port\_univ epdp\_key\_univ.

where epdp\_port\_univ (and the port type) are defined in UCBasicTypes.

Interested readers should study the prelude and smc-case-study files for further details and examples.

## UC DSL Interpreter Syntax

1. Preamble  
   The first two (non-comment) lines of a script file are

load UC-DSL-theory-name.

functionality real-func-expr.

where UC-DSL-theory-name is the name of a ".uc" file on the load path (but omitting ".uc") and real-func-expr is the name of a real functionality followed by a comma-separated list of arguments in parentheses for its parameters. If the real functionality has no parameters, the parentheses can be omitted. If the functionality is not defined in the loaded theory, it must be qualified with the theory name.

1. Variable declarations and assumptions

var var-name : type-expr.

assumption assum-name : bool-expr.

where var-name is a valid UC DSL identifier that begins with a lowercase letter, assum-name is a valid UC DSL identifier, type‑expr is a valid EasyCrypt type expression and bool-expr is a valid EasyCrypt expression of type bool. These may appear anywhere after the first two lines of the script file and are added to the global context. If they appear inside a session, they are removed when the session ends. The bool expression may refer to anything in the global context and may use the interpreter-unique envport operator with 2 arguments, typically

envport func port-expr.

1. Global SMT settings may be specified anywhere after the first two lines of the script file:

prover smt-info.

where smt-info is one or more of

* [prover-selector] or prover=[prover-selector]: a list of provers to run, as a list of strings in square brackets and separated by spaces
  + Recall easycrypt config lists known provers, among other things
  + Prover names can omit the version or specify all or part of it: "Alt-Ergo", "Alt-Ergo2" and "Alt-Ergo2.4.2" are all accepted, as of this writing
  + To modify a previous global selection, a string can be prefixed with + to add the prover to the list or - to remove it
* quorum=n: the minimum number of provers that must return success for the tactic to succeed
* timeout=n: the maximum time (in integer seconds) each prover has to respond
* maxprovers=n: the maximum number of provers to run in parallel
* verbose: set the verbosity level; also verbose=n, but n seems to have no effect
  + The tactic reports how long it ran
* selected: list the EasyCrypt axioms and lemmas that were sent to the SMT prover
* wantedlemmas=dbhint: select only the specified axioms and lemmas
  + A list of axioms and lemmas separated by spaces
  + An axiom or lemma can be excluded by prefixing it with – (or use unwantedlemmas)
  + All axioms and lemmas of theory T can be included (or excluded) using @T (or -@T)
* unwantedlemmas=dbhint: exclude the specified axioms, lemmas and theories
* all: select all available lemmas except unwantedlemmas and those marked nosmt
  + This option causes the wantedlemmas option to be ignored
* n or maxlemmas=n: the maximum number of lemmas that may be selected
  + This option may be broken in the current version
* iterate: incrementally augment the number of selected lemmas
* lazy: how this affects the smt tactic and/or SMT provers is unclear
* full: how this affects the smt tactic and/or SMT provers is unclear

1. Starting and ending a session

real.

ideal.

finish.

where real starts a session in the real world, ideal starts a session in the ideal world and finish ends the current session.

1. Sending messages

send source\_port @ message-path @ destination\_port.

send source\_addr $ message-path $ destination\_addr.

where source\_port and destination\_port are expressions of type port, which must be enclosed in parentheses unless they are variable names. If either port belongs to the functionality, you may instead provide only its address and use $ in place of @.

and where message-path has the form

theory-name . [ composite-interface-name . ] .

basic-interface-name . message-type (msg-args).

If there are no message arguments, the parentheses may be omitted.

1. Executing the functionality

step.

run.

where step executes the current statement of the active state machine or routes and delivers a message and run executes steps until the functionality yields control. These commands are also used after a null message transferring control between the environment and adversary.

The step command can optionally specify SMT settings:

step prover smt-info hint [ remove-lemmas / add-lemmas ]/.

where smt-info uses the syntax above for global SMT parameters and remove-lemmas and add-lemmas are both lists of qualified identifiers separated by commas. These settings take precedence over any global settings that are active but apply only to this step.

1. Asserting effects  
   The expected effect of a run or step command can be asserted immediately after with one of

assert rand.

assert fail\_out.

assert msg\_out sent-message entity-with-control.

assert ok.

where sent-message is the expected message and entity-with-control is either ctrl\_adv or ctrl\_env . It is a script error if the assertion is false.

1. Exiting the interpreter  
   The interpreter exits when it encounters the end of the script file or the command

quit.

# References

1. Ran Canetti, “Universally Composable Security: A New Paradigm for Cryptographic Protocols,” February 11, 2020, https://eprint.iacr.org/2000/067.pdf.
2. Ran Canetti, Alley Stoughton and Mayank Varia, “EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security,” of which there is a short version (2019 IEEE 32nd Computer Security Foundations Symposium (CSF)) with 17 pages) and an extended version (https://eprint.iacr.org/2019/582) with 43 pages.
3. EasyUC GitHub repository at https://github.com/easyuc/EasyUC.
4. EasyCrypt GitHub repository at https://github.com/EasyCrypt/easycrypt.
5. “EasyCrypt Reference Manual,” https://github.com/EasyCrypt/easycrypt-doc.
6. “EasyCrypt Guide for Programmers,” Riverside Research, 2022, unpublished.

1. Each vertical line is the *lifeline* of the *participant* indicated at the top. Time flows from top to bottom; left and right are not significant. Message transmission is considered instantaneous. Each time a participant receives a message, the box on its lifeline indicates it becomes active and can perform computations as needed. A participant goes inactive as soon as it sends a message. Note that, in these diagrams, sending a message does not correspond to calling a subroutine and there are no dashed "return" message lines. [↑](#footnote-ref-2)