Molly: A Verified Compiler for Cryptoprotocol Roles

Daniel J. Dougherty[1] Joshua D. Guttman[1,2]

[1]Worcester Polytechnic Institute
[2]The MITRE Corporation

Abstract

Molly is a program that compiles cryptographic protocol roles written in a high-level notation into straight-line programs in an intermediate-level imperative language, suitable for implementation in a conventional programming language.

We define a denotational semantics for protocol roles based on an axiomatization of the runtime. A notable feature of our approach is that we assume that encryption is randomized. Thus, at the runtime level we treat encryption as a relation rather than a function.

Molly is written in Coq, and generates a machine-checked proof that the procedure it constructs is correct with respect to the runtime semantics. Using Coq's extraction mechanism, one can build an efficient functional program for compilation.

Contents

1 Introduction

Protocol narrations, colloquially, "Alice & Bob" notation, are a nearly ubiquitous informal means to describe the intended executions of cryptographic protocols as sequences of cryptographic message exchanges among the protocol's participants. This notation is succinct and approachable but it lacks a formal semantics and leaves many implementation details implicit.

We are interested here in bridging the gap between informal description and implementation. To that end we present Molly, a compiler that transforms a narration in the cpsa input language into an implementation.

Molly generates executable code in an intermediate language to handle the transmissions and receptions dictated by the protocol, as well as any checks that must be done before taking one of these steps. For example if the protocol specifies reception of a message m, and at runtime the network delivers the value v_m, the code must

1. deconstruct v_m to verify that it has the structure required in the protocol specification, e.g. that it is a properly formed encryption using the expected key, and its plaintext is a tuple of components of the right kinds, and so on;
2. do a check that whenever some part of m is to be equal to some part of another message already processed, the corresponding runtime values are indeed equal,
3. store v_m and its components so that they can be used in the remainder of the execution as required.

In the dual case, when a message must be prepared and transmitted, the code must arrange to build up a suitably structured value from values already known, together with new values that may be chosen randomly, such as nonces and session keys.

In addition, we define a denotational semantics for protocol roles given in the cpsa language, defined in terms of "transcripts", runtime traces of messages as bitstrings. Our intermediate language will be seen to have an obvious transcript semantics, and our main theorem is a correctness theorem relating role transcripts and transcripts for our intermediate language.

Molly is written in Coq's Gallina programming language, and our correctness result is developed and checked within Coq. Our Coq development is available at `github.com/dandougherty/Molly`

1.1 Outline and Contributions

There were a number of key goals that shaped our work reported here.

Code Generation for an Intermediate Language. Functionally, MOLLY is a partial function from roles to *procs*; the latter are essentially the Procs of Ramsdell's Roletran [Ram21]. A proc is a sequence of intermediate-level instructions. Procs are straightforward to translate into concrete executable programming languages, but also easy to characterize semantically. Hence, they are an attractive target language for MOLLY as a verified compiler. MOLLY is written in Coq's Gallina specification language, so that using Coq's extraction mechanism, we derive code, in Ocaml for instance, to produce a proc from a role.[1]

Transcripts for Roles. The crucial first step in proving correctness of this compilation is defining the notion of *transcript for a role:* a transcript is a sequence of runtime values which might arise as a runtime execution of the role (Section 8.2). Transcripts live in a domain—bitstrings—independent of symbolic messages. It seems suitable to say that role transcripts yield a *denotational semantics* for protocol roles: the meaning of a role is the set of its transcripts. Put another way, the meaning of a role is its set of *observable actions*.

The development of this idea is somewhat subtle, chiefly because we take randomized encryption seriously. We feel that our development of transcripts gives useful perspective on protocols even outside the context of compiling, and we count it as one of our main contributions. We also define the notion of *transcript for a proc* (Section 8.3): this definition is utterly straightforward.

Take note of the distinction between a semantics for individual roles, treated here, and a semantics for *protocol executions*, which of course comprise interactions between individual role executions. The former, our transcripts, are merely "sections" of the latter, the behaviors observable by a single principal.

We want to stress the point that without a semantics for roles and procs defined independently from each other, it doesn't even make sense to claim that a compilation is correct, much less prove such a claim

Treatment of Randomized Encryption. Manipulating symbolic terms is useful only if we maintain a realistic view of the relationship between terms and actual network messages consisting of bitstrings. Because many cryptographic operations are randomized, the bitstrings are not in one-to-one correlation with the symbolic terms—even given an assignment of bitstrings to the variables occurring in the terms. In particular calling an encryption or digital signature primitive on the same message twice, with the same key, does not yield the same bitstring result.

This has the semantic consequence that role transcripts are generated from *relations* from values to bitstrings (the "valuations" of Section 8.2.1). In turn, our compiler embodies choices about which bitstring should be used as the representative of a symbolic term when more than one is available, and our correctness proof characterizes the constraints on these strategies.

[1] It is straightforward to translate procs into code for a conventional language with a library for cryptographic primitives such as C, Rust, or Java; we have not pursued this at this point in the project.

Axiomatizing the Runtime. Our correctness theorem is about transcripts; transcripts are sequences of runtime values, so we require some analysis of the runtime. A key feature of our approach is that we do not *define* a runtime for our proof, rather we simply isolate surprisingly few mild assumptions about the runtime we require for the proof. This strategy has the obvious benefits of broadening the applicability of the result and of identifying the principles that make compilation correct.

A Machine-Checked Correctness Theorem. Our main theorem, the Reflecting Transcripts theorem (Theorem 9.1), states that if role rl is compiled to procedure pr, any transcript for pr is a transcript for rl. As explained in Section 10.1, the converse of Reflecting Transcripts theorem cannot be expected to hold in the presence of randomized encryption.

Proof-Theoretic View of Code Generation. We have organized our compilation according to the Dolev-Yao model [DY83], by which we mean that we view the principal executing a role as manipulating term-structured items according to derivation rules. Whatever may be said about the strengths and weaknesses of the Dolev-Yao model as a model of the adversary, this is certainly a reasonable model of the regular, compliant protocol participants; the protocol designer does not expect them to have to succeed at cryptanalysis, for instance, to run the protocol successfully. Thus, reflecting much previous work, dating back at least to Paulson and Marrero et al. in the 1990s [Pau97, CJM98], we take a proof-theoretic view of the actions of our compiler. It emits code by executing steps that we formalize as inference rules, thus generating derivations in a Gentzen-Prawitz natural deduction calculus [Gen69, Pra65]. Details can be found in Section 7.2.

Compilability and Executability. Several authors have given definitions of "executability" of a role, intending to capture the informal notion of a role providing enough information to be able to run (notably, having decryption keys available when required). Existing definitions tend to be incomparable across formalisms. The results of Section 7.2 allow us to characterize the input roles on which compilation *fails*. And this in turn allows us to motivate and define a notion of *executability* for a role in terms of the well-known notion of Dolev-Yao derivability.

1.2 Related Work

There is growing interest in the development of verified compilers, for conventional languages (*e.g.*, [Ler09]) as well as domain-specific languages (*e.g.*, [PGLSN22]). To our knowledge Molly is the first compiler for cryptographic protocols with a machine-checked correctness condition.

Many authors have worked to bridge the gap between protocol narrations in a semi-formal style and protocol descriptions that are more formal. We organize the discussion below according a crude partition. One category is work

that translates protocol narrations to another—formally defined—language, like process calculus or multiset rewriting. Typically the payoff is that automated verification tools can then be used to reason about the protocol. Another category is tools that compile protocol descriptions into a conventional programming language. Often the input to these tools is already in a formal notation such as spi-calculus, and sometimes the translation is instrumented with tools that support claims about the security guarantees of the target program.

Work in the first category includes translation of protocol narration to CSP [LBH97], to generic intermediate languages [DM00] [Möd09], [AMV15], to Multiset Rewriting [JRV00] [KB14], to symbolic representations of principals' knowledge [MK08] [BKRS15] (annotated with unifiability conditions in [CR10]), and to Pi-calculus and variants [CVB05] [CVB06] [BN07].

In most of the works above the authors offer their work as supporting an "operational semantics" for roles, and here we can identify an interesting difference between our work and theirs. In the works above we can identify two broad operational semantics approaches. In one case the job is to define the *activities that an agent takes* to implement the protocol: constructing and deconstructing messages, certain checks, etc. In the other case one defines the possible *executions of the protocol:* sometimes a notion of trace is defined, tracking the evolution of symbolic representation of principal's knowledge, (Cremers [CM05] is a detailed development of this perspective) or alternatively the semantics is implicit in the semantics of the target formalism (pi-calculus, multiset rewriting, etc).

Our work cuts across these two functions. The sequence of activities that an agent takes to implement a given protocol role is precisely the proc built by our compilation. And, our transcripts capture the executions of the protocol not in terms of symbolic terms, but rather in terms of bitstrings.

We will see that our procs support an obvious notion of transcript. Then as noted above, since roles and procs have a common target domain for their semantics it makes sense to compare the meaning of a role and the meaning of a proc. Prior work offers no formal proof of correctness of the translation process; our main contribution is a machine-checked proof of a theorem doing just that.

An intriguing aspect of the work in Caleiro, Vigano, and Basin [CVB06] is that they employ a notion of incremental symbolic runs as a basis for a *denotational* semantics. Each state of a run reflects the information known by the principals; the run itself models how information grows. The rules for evolution of these runs look very much like our saturation process in Section 6 for building the bindings of a proc! The difference is that for them the process of recording the growth of principals' knowledge *is* a semantics of a role, while for us this process is the essence of *compiling*, not execution. As explained earlier, our denotational semantics is grounded in the world of bitstrings.

Arquint et al [AWL+22], [AWL+23] are mainly interested in verification, and present a tool that is not really a protocol compiler: it starts with a Tamarin

model and generates a set of *I/O specifications* in separation logic. But their main correctness result has an interesting relationship to ours. An I/O specification is a set of permissions needed to execute an I/O operation [PJP15]. Then (quoting [AWL+22]) "traces can intuitively be seen as the sequences of I/O permissions consumed by possible executions of the programs that satisfy it." They prove that if abstract Tamarin model M is translated to a set S of I/O specifications, then any concrete implementation satisfying S refines M in terms of trace inclusion. This is closely analogous to our Reflecting Transcripts theorem 9.1, with logical properties standing in for bitstrings.

We now turn to the category of projects that are primarily focused on generating implementations from protocol specifications. Although the work in [AMV15] is not principally focused in this way, that paper reports a translation from its intermediate language SPS into JavaScript.

Tobler and Hutchinson [TH05] built the Spi2Java tool, which builds a Java code implementation of a protocol specified in a variation of the Spi calculus. There is no proof that the semantics of the input specification is preserved by the translation.

Backes, Busenius, and Hritcu [BBH12] developed Expi2Java, which translates models written in the Spi calculus [AG97] into Java. They formalized their translation algorithm in Coq and proved that the generated programs are well-typed if the original models are well-typed.

Modesti [Mod14, Mod16] developed the "AnBx" compiler, which generates Java code from protocols written in an Alice & Bob-style notation. The tool generates certain consistency checks and annotates the translation with applied pi-calculus expressions to permit a ProVerif [Bla16] verification that security goals are met by the Java code.

The JavaSPI tool of Sisto, Copet, Avalle and Bronte [SBCAP18] starts with code in a fragment of the language that corresponds to applied pi-calculus [ABF17]. The tool can symbolically execute this code in the Java debugger, formally verify it using ProVerif, eventually refine to an Java implementation of the protocol. They prove that a simulation relation relates the Java refined implementation to the symbolic model verified by ProVerif.

Spi2Java, Expi2Java and JavaSPI require the user to provide input in a more demanding formalism than the familiar Alice & Bob-style. A benefit of all of the systems in this category compared with Molly is the fact that they produce code for the ubiquitous Java platform. On the other hand, for none of these systems is there a proof of correctness of the compilations themselves.

Ramsdell's Roletran compiler [Ram21] has functionality and overall goals quite close to that of Molly. The input is a CPSA specification of a role, and the output is a program in an intermediate language designed to be readily translated to a conventional language: our input and output languages are inessential variations on Roletran's. The main correctness claim of Roletran is that "the procedure produced by Roletran is faithful to [the] strand space semantics [of

the input role]." Roletran does not have a machine-checked proof of its global correctness claim, but the distribution does the following interesting thing. Coq scripts are provided that can check, for a given role rl, that the procedure generated by the tool is correct (according to the symbolic-trace semantics).

Differences between MOLLY and Roletran include the facts that Roletran's strand space semantics is in terms of *symbolic traces* for a role, as opposed to our role semantics based on bitstrings, and that we provide a machine-checked proof of a uniform correctness theorem. Roletran has some restrictions on the messages that can appear in roles compared with MOLLY.

Roletran is also the inspiration for a fully usable framework called *Zappa* that augments a role compiler with many ingredients needed for a runtime system, including runtime message formats based on ASN.1 encodings. The Zappa compiler generates procedures in the Rust programming language for a substantial extension of the source syntax considered here and in Roletran. Correctness proofs have not been considered for the extensions. Cryptographic libraries available in Rust may be linked in.

1.3 Road map

We introduce the main ideas of compilation, and the notation for roles and procs, through a series of examples in Section 2. Section 3 gives preliminary definitions, and Section 4 lists our assumptions about the runtime. Section 5 explains the compiler at a high level; section 6 presents the details of the algorithm. Section 7 presents some results about the procs constructed by the compiler. Section 8 defines our formal models of execution of a role and of a proc, and section 9 proves our main theorem, the Reflecting Transcripts theorem.

2 Overview and Examples

The input to MOLLY is a description of one role of a cryptographic protocol. There are a variety of notations to specify protocols; here we use the input language of the protocol analysis tool CPSA. The output of MOLLY is a program in an intermediate language which is readily translatable to a program in a language such as C or Rust. A program in this intermediate language will be called a *proc*. Roles and procs are defined precisely in Sections 3.3 and 3.4.

Our execution model is that procs read and write runtime values on channels, maintain a local store of runtime values, and execute commands that update their state.

To provide a little more detail into the way a role specification is translated to proc code, and to introduce some of the subtleties that arise in this translation, we discuss a series of small examples.

Example 1. Here is a simple role. It takes three parameters, a channel and two atoms of sort Data. It sends over the channel an ordered pair constructed from the data, and subsequently expects to receive one of the data items over the same channel.

(Prm (Ch 1));
(Prm (Dt 1));
(Prm (Dt 2));
(Snd (Ch 1) (Pr (Dt 1) (Dt 2)));
(Rcv (Ch 1) (Dt 2))

Below is a sequence of instructions in our proc language to execute this role. (Line numbers are added to the display of the proc here for facilitate commentary.)

For each parameter, a location is allocated, the appropriate value is stored; and a runtime check is emitted to ensure that the parameter value has the expected sort (for example, lines 2–4 for the first parameter).

For the transmission: we create a location to store this pair-value (line 14) and send the value in this location over channel 1 (line 15).

For the reception: the reception of data from channel 1 and implicit declaration of the location for the data are recorded (line 17), the data stored in that location (line 18), and a check is done that the incoming value is the same as the expected one (line 19).

Proc 1: for Example 1

```
1       (* first parameter *)
2       Evnt (Prm (L 1));
3       Bind (Ch 1, L 1) (Param 1);
4       Csrt (L 1) Chan;
5       (* second parameter *)
6       Evnt (Prm (L 2));
7       Bind (Dt 1, L 2) (Param 2);
8       Csrt (L 2) Data;
9       (* third parameter *)
10      Evnt (Prm (L 3));
11      Bind (Dt 2, L 3) (Param 3);
12      Csrt (L 3) Data;
13      (* the send action *)
14      Bind (Pr (Dt 1) (Dt 2), L 4) (Pair (L 2) (L 3));

15      Evnt (Snd (L 1) (L 4));
16      (* the receive action *)
17      Evnt (Rcv (L 1) (L 5));
18      Bind (Dt 2, L 5) (Read 1);
```

```
19      Same (L 5) (L 3)
```

///

Example 2. If we view role init1 as initiating a session, the following is a role that could serve as responder.

It takes one parameter, the channel. The two parameters from role init1 are now (expected to be) components of the single value received by resp1. Assuming the reception is accepted, the second component is sent back on channel 2.

(Prm (Ch 1));
(Rcv (Ch 1) (Pr (Dt 1) (Dt 2)));
(Snd (Ch 1) (Dt 2))

Below is the proc generated by the compiler. The parameter (Ch 1) is processed just as before. The reception of ((Pr (Dt 1) (Dt 2)) begins with the binding of the receive runtime value to location L 2. We next generate code that will check whether we have received a suitable value: indeed the actual value received might not be a pair at all, or it might be a pair of values not of sort Data.

We deconstruct the received value by binding locations to the result of operators Frst and Scnd (lines 7 and 8). Crucially, these operators can fail at runtime: the expression Frst fails if given a non-pair as input, and otherwise returns the first component of the pair; similarly for Scnd.

Once those deconstructions are done we expect to be left with atomic values; we emit the appropriate sort-check statements; these will of course fail on input of the wrong sort.

For the Scnd transaction: since the value to be sent will already available in location 4, we simply emit the Snd event statement.

Proc 2: for Example 2

```
1       Evnt (Prm (L 1));
2       Bind (Ch 2, L 1) (Param 1);
3       Csrt (L 1) Chan;
4
5       Evnt (Rcv (L 1) (L 2));
6       Bind (Pr (Dt 1) (Dt 2), L 2) (Read 1);
7       Bind (Dt 1, L 3) (Frst (L 2));
8       Bind (Dt 2, L 4) (Scnd (L 2));
9       Csrt (L 4) Data;
10      Csrt (L 3) Data;
11
12      Evnt (Snd (L 1) (L 4))
```

///

Example 3.

(Prm (Ch 1));
(Snd (Ch 1) (Dt 1));
(Rcv (Ch 1) (Hs (Dt 1)))

There are two new things to talk about here.

First, we want to send a value corresponding to ((Dt 1)), which is a value we do not have stored in advance. Our procs have an operator Genr that produces new data of a specified sort (line 2). In our proc language this operator also takes a natural number parameter, to achieve referential transparency: different occurrences of a Genr expression always denote the same value. When procs are translated to stateful languages like C or Rust, this parameter might disappear.

As a programming design decision, MOLLY emits all the necessary Genr statements at the start of the code generation process. Thus the code for generation of (Dt 1) appears even before the processing of parameters.

Second, we need to ensure that the value received for the ((Hs (Dt 1))) term really is the hash of the value received for the ((Dt 1)). But Hs has no analogue to Frst or Scnd in our proc language of course: it is computationally infeasible to destruct the hash operator. Instead, we rely on the fact that we do have the value for ((Dt 1)) stored, the one we generated for the transmission. We compute the hash of *that* value (line 15) and compare that to the value received (line 16).

Proc 3: for Example 3

```
1      (* allocation for fresh value *)
2      Bind (Dt 1, L 1)  (Genr 1 Data);
3
4      (* first parameter *)
5      Evnt (Prm (L 3));
6      Bind (Ch 1, L 3) (Param 1);
7      Csrt (L 3) Chan;
8
9      (* transmission *)
10     Evnt (Snd (L 3) (L 1));
11
12     (* reception *)
13     Evnt (Rcv (L 3) (L 4));
14     Bind (Hs (Dt 1), L 4) (Read 1);
15     Bind (Hs (Dt 1), L 2) (Hash (L 1));
16     Csame (L 4) (L 2)
```

///

Example 4. Here is a role that our compiler will decline to compile.

```
(Prm (Ch 1));
(Rcv (Ch 1) (Hs (Dt 1)));
(Snd (Ch 1) (Dt 1))
```

Since our proc does not have any stored value corresponding to (Dt 1) at the time of the reception, we cannot emit any code to verify at runtime that the received value is a correct hash. It is presumably possible to check that the value received is a hash of *something,* so it might be tempting to call that something "(Dt 1)" but we certainly could not use the term (Dt 1) elsewhere in the role since we have no information about its value. ///

In Example 4 one might be tempted to expect that MOLLY would use Genr to generate a location bound to (Dt 1) initially, so that we could use the same strategy as there (computing the hash of the known value and comparing equality). But the operator Genr is only performed on locations corresponding to terms that *originate* in the role, that is, terms whose first occurrence in the role is part of a transmission. The notion of origination is a central concept in the analysis of protocols by CPSA.

Public-Key Encryption

Example 5.

```
(Prm (Ch 1)); (Prm (Ik (Av 2)));
(Rcv (Ch 1) (En (Nm 0) (Ak (Av 2))));
(Snd (Ch 1) (Nm 0))
```

The term (Ak (Av 2)) denotes an asymmetric key generated from the variable (Av 2). By convention this denotes the public key associated with this variable. The term (Ik (Av 2)) denotes the corresponding private key; these two terms make a *key pair*.

The fact that (Ik (Av 2)) is a parameter reflects the idea that this key is known to the principal executing this role; the role can decrypt an incoming messages encrypted with the key-partner of this term. This is precisely what happens in line 11: the encryption is stored in location ($L3$) and the decryption key is expected to be stored in location ($L2$), we allocate a new location ($L4$) and store the result of the decryption there (line 11).

It is important to note that just as with Frst and Scnd, a Decr operation can fail. That is, if the runtime values appearing at a reception do not fit the specification embodied in the role, execution will halt.

Proc 4: for Example 5

```
 1      Evnt (Prm (L 1));
 2      Bind (Ch 1, L 1) (Param 1);
 3      Csrt (L 1) Chan;
 4
 5      Evnt (Prm (L 2));
 6      Bind (Ik (Av 2), L 2) (Param 2);
 7      Csrt (L 2) Ikey;
 8
 9      Evnt (Rcv (L 1) (L 3));
10      Bind (En (Nm 0) (Ak (Av 2)), L 3) (Read 1);
11      Bind (Nm 0, L 4) (Decr (L 3) (L 2));
12      Csrt (L 4) Name;
13
14      Evnt (Snd (L 1) (L 4))
```

///

Example 6 (Encryption 2)**.**

The following role will be rejected by the compiler.

(Prm (Ch 1));
(Rcv (Ch 1) (En (Nm 0) (Ak (Av 2))));
(Snd (Ch 1) (Nm 0))

There is no way to decrypt the received message, since the key partner of (Ak (Av 2)) is not a parameter nor can it be constructed from the data available at the time of the reception. ///

Treating Randomized Encryption

Example 7. As a simple first example for symmetric encryption consider a role that expects an encryption whose key is the hash of a known value and replies with the decrypted plaintext.

(Prm (Ch 1)); (Prm (Dt 2));
(Rcv (Ch 1) (En (Nm 0) (Hs (Dt 2))));
(Snd (Ch 1) (Nm 0))

A suitable proc follows. For readability we introduce some whitespace in the listing to make it easier to focus on the treatment of encryption.

Here

- `(L 1)` stores the value of the channel, and `(L 2)` stores the value of (Dt 2)

- In line 9 we construct the hash of (Dt 2) in preparation for constructing and storing it in `(L 3)`
- the received value is stored in `(L 4)`.
- In line 16 we decrypt the received value with the key we constructed in line 9. If this decryption succeeds (because the reception was indeed encrypted with the value of (Hs (Dt 2)) we constructed) we store the result in `(L 5)` and subsequently send it.

Proc 5: for Example 7

```
1       Comm "input (Ch 1)";
2       Evnt (Prm (L 1));
3       Bind (Ch 1, L 1) (Param 1);
4       Csrt (L 1) Chan;
5       Comm "input (Dt 2)";
6       Evnt (Prm (L 2));
7       Bind (Dt 2, L 2) (Param 2);
8
9       Bind (Hs (Dt 2), L 3) (Hash (L 2));
10
11      Csrt (L 2) Data;
12      Comm "receiving (En (Nm 0) (Hs (Dt 2)) ) on (Ch
            1)";
13      Evnt (Rcv (L 1) (L 4));
14
15      Bind (En (Nm 0) (Hs (Dt 2)), L 4) (Read 1);
16      Bind (Nm 0, L 5) (Decr (L 4) (L 3));
17
18      Csrt (L 5) Name;
19      Comm "sending (Nm 0) on (Ch 1)";
20      Evnt (Snd (L 1) (L 5))
```

///

In the symbolic model messages are represented as elements of a free algebra for an equational theory, the *message algebra*. In particular encryption is named by a term (En m k).

But we assume that our runtime implements *randomized* symmetric encryption.[2]

Thus we have a mismatch between the traditional algebraic semantics of term algebras and the runtime semantics.

[2]We do not attempt to *model* randomized encryption, that is, we do not formalize any mechanisms in the runtime or in the term algebra that reflect computational processes of encryption and decryption that use randomization.

For example, if a term (En m k) occurs more than once in a protocol description it may be considered to denote different runtime values at different occurrences.

So the problem arises of how to define a semantics for the message algebra that is faithful to runtime semantics. (Say that we aren't going to try to model the probabilistic aspect, just the possibilistic aspects.)

By the way, these considerations arise prior to the idea of writing a compiler: they are just about the establishing the meanings of cryptographic terms.

Here is a series of examples that highlight some ways in which randomized encryption poses interesting design choices for our compiler.

Example 8 (A simple failure)**.** The following role will be rejected by the compiler, for the same reason as in Example 6

(Prm (Ch 1));
(Rcv (Ch 1) (En (Dt 3) (En (Dt 1) (Dt 2))))

There is have no way to decrypt the received message, since we cannot construct (En (Dt 1) (Dt 2)) as a decryption key from the data available. ///

Compare the next example with Example 7.

Example 9.

(Prm (Ch 1)); (Prm (Dt 1)); (Prm (Dt 2));
(Rcv (Ch 1) (En (Dt 3) (En (Dt 1) (Dt 2))))

This role can be compiled, since we can construct (En (Dt 1) (Dt 2)). But the resulting code has negligible probability of executing successfully: the runtime value of (En (Dt 1) (Dt 2)) we construct from the local data (to be used as decryption key) is unlikely to be the runtime value of (En (Dt 1) (Dt 2)) that was used by a peer as the encryption key to form (En (Dt 3) (En (Dt 1) (Dt 2))).

///

Another example, similar to the previous but with yet another subtlety.

Example 10.

(Prm (Ch 1)); (Prm (Dt 1)); (Prm (Dt 2));
(Snd (Ch 1) (En (Dt 1) (Dt 2)));
(Rcv (Ch 1) (En (Dt 3) (En (Dt 1) (Dt 2))))

This role can be compiled, since we can (and did!) construct (En (Dt 1) (Dt 2))

Under a strictly mathematical analysis the likelihood of successful execution here is the same as in the previous example. But it is plausible to imagine that the received message would instantiated as the same runtime value for (En (Dt 1) (Dt 2)) as was sent in the second message, for example if this were the conventional understanding of the protocol. But the protocol specification language provides no way to express this convention. ///

Section 7.3 explains why this phenomenon is, in a sense, inevitable.

3 Preliminaries

3.1 Actions

The Act parameterized data type is a useful device for tying together several recurring constructions. The constructor Prm builds parameters, Ret builds return values, and Rcv and Snd builds values received and sent.

We will shortly define roles, procs, and runtime with respective carriers Terms, Locations and runtime values. Expressions of type (Act Term) will denote symbolic terms as parameters, sent messages, etc; expressions of type (Act Loc) will denote locations where parameters, sent messages, etc are stored; expressions of type (Act Rtval) will denote runtime values used as parameters, sent messages, etc.

Having a polymorphic datatype to refer to corresponding construction allows a uniform treatment of important correspondences.

Definition 3.1. If X is a Type, $Act\ X$ is the type whose constructors are

$$\begin{aligned}\mathsf{Prm} &: X \to Act\ X\\ \mathsf{Ret} &: X \to Act\ X\\ \mathsf{Rcv} &: X \to X \to Act\ X\\ \mathsf{Snd} &: X \to X \to Act\ X\end{aligned}$$

3.1.1 Mapping

Map a function or a relation over Act.

Definition 3.2 (Mapping over Act)**.** Let $r : X \to 2^Y$ be a relation from X to Y. The relation r^{Act} from $(\mathsf{Act}\ X)$ to $(\mathsf{Act}\ Y)$ is the natural extension of r to $(\mathsf{Act}\ X)$:

$$\begin{aligned}r^{\mathsf{Act}}((\mathsf{Prm}\ x)\ (\mathsf{Prm}\ y)) &\text{ holds if } (r\ x\ y)\\ r^{\mathsf{Act}}((\mathsf{Ret}\ x)\ (\mathsf{Ret}\ y)) &\text{ holds if } (r\ x\ y)\\ r^{\mathsf{Act}}((\mathsf{Rcv}\ x_1\ x_2)\ (\mathsf{Rcv}\ y_1\ y_2)) &\text{ holds if } (r\ x_1\ y_1) \text{ and } (r\ x_2\ y_2)\\ r^{\mathsf{Act}}((\mathsf{Snd}\ x_1\ x_2)\ (\mathsf{Snd}\ y_1\ y_2)) &\text{ holds if } (r\ x_1\ y_1) \text{ and } (r\ x_2\ y_2)\end{aligned}$$

As a special case (modulo notation) we observe that if $f : X \to Y$ then we have the function $f^{\mathsf{Act}} : (\mathsf{Act}\ X) \to (\mathsf{Act}\ Y)$ with

$$f^{\mathsf{Act}}((\mathsf{Prm}\ x)) = (\mathsf{Prm}\ (fx))$$

$$f^{\mathsf{Act}}((\mathsf{Ret}\ x)) = (\mathsf{Ret}\ (fx))$$
$$f^{\mathsf{Act}}((\mathsf{Rcv}\ x_1\ x_2)) = (\mathsf{Rcv}\ (fx_1)\ (fx_2))$$
$$f^{\mathsf{Act}}((\mathsf{Snd}\ x_1\ x_2)) = (\mathsf{Snd}\ (fx_1)\ (fx_2))$$

Definition 3.3 (Mapping over a list)**.** Let r be a relation from X to Y. The relation $\mathrm{map_R}\, r$ is the relation from lists of X to lists of Y defined by:

$$\mathrm{map}\, r\ [x_1, \ldots x_n]\ [y_1, \ldots y_n] \text{ if } \forall\ 1 \leq i \leq n,\ (r\ x_i\ y_i)$$

Of course when r is a function $\mathrm{map_R}$ is the standard "map" operation mapping a function over a list.

Lemma 3.4. *If r and r' are relations from X to Y with $r \subseteq r'$ then for all lists xs and ys, $(\mathrm{map_R}\, r\ xsys)$ implies $(\mathrm{map_R}\, r'\ xs\ ys)$.*

Proof. easy. ///

Notation 3.5. If r_{XY} is a relation from X to Y and r_{YZ} is a relation from Y to Z then $(r_{XY}; r_{YZ})$ denotes the relational composition of r_{XY} with r_{YZ}, a relation from X to Z.

Lemma 3.6. *If r_{XY} is a relation from X to Y and r_{YZ} is a relation from Y to Z then*

$$(r_{XY}; r_{YZ})^{\mathsf{Act}} = {r_{XY}}^{\mathsf{Act}}\ ;\ {r_{YZ}}^{\mathsf{Act}}$$

Proof. easy. ///

3.2 Sorts

Symbolic terms, proc expressions, and runtime values obey a common sort discipline.

Definition 3.7. The *sorts* are

chan **data** **name** **text** **skey** **akey** **ikey** **mesg**

The *base* sorts are the sorts other than **mesg**.

3.3 Terms and Roles

We begin with a set of atoms, each of which has a sort. We close under the operations of pairing, encryption, hashing, and quotation.

It is convenient to define raw symmetric and asymmetric keys as auxilliary notions, which will be wrapped to form terms.

$$\text{skeys} \stackrel{\text{def}}{=} \{(\mathsf{Sv}\ n) \mid n \in \mathbb{N}\}$$
$$\text{akeys} \stackrel{\text{def}}{=} \{(\mathsf{Av}\ n) \mid n \in \mathbb{N}\}$$

The set of terms is defined inductively by the following constructors.

$\mathsf{Ch} : \mathbb{N} \to \text{Term}$	channels
$\mathsf{Tx} : \mathbb{N} \to \text{Term}$	text
$\mathsf{Dt} : \mathbb{N} \to \text{Term}$	data
$\mathsf{Nm} : \mathbb{N} \to \text{Term}$	names
$\mathsf{Sk} : \text{skeys} \to \text{Term}$	symmetric keys
$\mathsf{Ak} : \text{akeys} \to \text{Term}$	asymmetric keys
$\mathsf{Ik} : \text{akeys} \to \text{Term}$	asymmetric keys
$\mathsf{Qt} : \text{string} \to \text{Term}$	quotation
$\mathsf{Pr} : \text{Term} \to \text{Term} \to \text{Term}$	pairs
$\mathsf{En} : \text{Term} \to \text{Term} \to \text{Term}$	encryptions
$\mathsf{Hs} : \text{Term} \to \text{Term}$	hashes
$\mathsf{Mg} : \mathbb{N} \to \text{Term}$	generic variable

The *elementary* terms are the terms *other than* those whose top-level constructor is one of Pr, En, Hs, or Qt.

A *symbolic key pair* is an ordered pair consisting of a private key and a public key (in that order) which are inverses for asymmetric encryption. Syntactically a symbolic key pair takes the form

$$((\mathsf{Ik}\ (\mathsf{Av}\ n)), (\mathsf{Ak}\ (\mathsf{Av}\ n)))$$

Either of the two parts uniquely determines the other, as evidenced by the symbolic syntax, but when terms are interpreted by runtime values, the public can be feasibly computed from the private part, but not vice-versa.

3.3.1 Sorts for Terms

Definition 3.8. Each term is assigned a sort.

$\text{sort}(\mathsf{Ch}\ x) = \textbf{chan}$	$\text{sort}(\mathsf{Tx}\ x) = \textbf{text}$
$\text{sort}(\mathsf{Dt}\ x) = \textbf{data}$	$\text{sort}(\mathsf{Nm}\ x) = \textbf{name}$
$\text{sort}(\mathsf{Sk}\ x) = \textbf{skey}$	$\text{sort}(\mathsf{Ak}\ x) = \textbf{akey}$
$\text{sort}(\mathsf{Ik}\ x) = \textbf{ikey}$	otherwise: $\text{sort}\, t = \textbf{mesg}$

3.3.2 Term Inverse

The inverse function t^{-1} on algebraic terms t is not named by a constructor, but it is definable.

Definition 3.9. The *inverse* of a term t, denoted t^{-1} is defined as follows.

- If (t_1, t_2) is a symbolic key pair then ${t_1}^{-1} = t_2$ and ${t_2}^{-1} = t_1$
- $t^{-1} = t$ for all t not part of a key pair

3.3.3 Roles

A role of a protocol will specify the parameters, the messages to be sent and received, and the outputs. Our roles are essentially the roles of CPSA, though in CPSA a role description also defines which terms are to be assumed to be "uniquely originating." But this information—crucial to *analysis*—is not relevant to compiling. If we omit this aspect of CPSA role specifications the constructors of the type Act capture everything we need about a role, leading to the following simple definition.

Definition 3.10 (Role)**.** A role is a list of (Act Term).

3.4 Procs

Procs are an intermediate language for representing straight-line cryptographic programs. It will be clear that a proc can be readily translated—with the help of a suitable cryptographic library—into a conventional imperative program.

We start with a set **Loc** of *locations* for storing runtime values.

Expressions An *Expression* is built from locations using certain operators that mirror the runtime operators presented in Section 4.

(Pair v_1 v_2) :	pairing values
(Frst v) :	first projection out of a pair
(Scnd v) :	second projection out of a pair
(Encr v_1 v_2) :	encryption
(Decr v_1 v_2) :	decryption
(Hash v) :	hashing a value
(Quot s) :	a string constant
(PubOf v) :	public-key partner of a private key
(Genr n srt) :	the nth value generated (to be of sort srt)
(Param n) :	the nth input parameter
(Read n) :	the nth value read from any channel

Statements A proc is a sequence of *statements*. Each statement is an *Event*, a *Bind*, or a *Check*.

1. An Event is one of the following four forms
   (a) (Prm v) : an input parameter is stored in location v
   (b) (Ret v) : the value in location vis output
   (c) (Rcv v_1 v_2) : when the value in v_1 is a channel, the value received on v_1 is stored in location v_2
   (d) (Snd v_1 v_2) : when the value in v_1 is a channel, the value stored v_2 is sent on location v_1
2. A Bind is of the form
   $$\mathsf{Bind}\ (t, v)\ e$$
   where t is a symbolic term, v is a location, and e is an expression
   This is an assignment statement, storing the value named by e into the location v; the symbolic term t serves as a type for the location v.
3. A Check is one of the following statement forms. A check is an assertion: if it succeeds, computation simply continues, and if it fails, computation halts.
   (a) (CSrt v s) : checks that the value in v_1 has the sort s
   (b) (CSame v_1 v_2) : checks that the values in the given locations are the same
   (c) (CKypr v_1 v_2) : checks that the values in the two locs make a private/public runtime key pair
   (d) (CHash v_1 v_2) : checks that the value in v_2 is the hash of the value in v_1
   (e) (CQot v s) : checks that the value in v is the string s

4 Axiomatizing the Runtime

Molly does not read or generate runtime expressions, but the semantics of roles and procs is built on runtime values. Here we record our assumptions about the runtime as an axiomatic theory $\mathcal{R}$. The theorems we prove about Molly will hold about any implementation of the runtime below which qualifies as a model of $\mathcal{R}$.

4.1 The Runtime Operators

Notation 4.1.

- We indicate that a function is partial by writing its return type as a lifted type (for example the return type of the operator `frst` below).
- When writing about equalities between expressions involving partial functions in this document we use the following convention:

$$e_1 \downarrow e_2$$

  asserts that both e_1 and e_2 denote and their values are equal.

Definition 4.2. The signature for theory $\mathcal{R}$ is

`pair` : Rtval → Rtval → Rtval	pairing
`frst` : Rtval → $\text{Rtval}_\perp$	first projection
`scnd` : Rtval → $\text{Rtval}_\perp$	second projection
`encr` : Rtval → Rtval → Rtval → bool	encryption
`decr` : Rtval → Rtval → $\text{Rtval}_\perp$	decryption
`hash` : Rtval → Rtval	hashing
`quot` : string → Rtval	quotation
`pubof` : Rtval → $\text{Rtval}_\perp$	public key partner for private key
`gen` : $\mathbb{N}$ → Sort → Rtval	value generation
`rtsort` : Rtval → Sort	check sort

We have not included here any operators for processing parameters to a role or reading values from a channel since we don't analyze these processes.

4.1.1 Key Pairs and Runtime Inverse

An ordered pair (r_1, r_2) is a *key pair* if it comprises the private and public parts of an asymmetric key, that is, if `pubof` $r_1 = r_2$. We use the name `kypr` for this defined relation:

$$\texttt{kypr}\ r_1\ r_2 \quad \text{if and only if} \quad \texttt{pubof}\ r_1 = r_2.$$

The following relation `rtinv` is not a runtime primitive, it is a definable relation convenient for analysis.

Definition 4.3. The *runtime inverse* `rtinv` is another definable relation, given by

$$\begin{array}{rl} (\texttt{rtinv}\, r_1\ r_2) & \text{if } (r_1, r_2) \text{ or } (r_2, r_1) \text{ make a runtime key pair} \\ (\texttt{rtinv}\, r\ r) & \text{if } r \text{ is not of sort } \textbf{akey} \text{ or } \textbf{ikey} \end{array}$$

It is easy to see that the relation `rtinv` is actually a function: for each r there is a unique r' such that $(\texttt{rtinv}\ r\ r')$. But this function is not feasibly implementable, under the assumption that one cannot feasibly compute the private part of a key pair from the public part. So we prefer to make our core definitions and axioms below in terms of `rtinv` as a relation: it is clearly *is* feasibly implementable under the assumption that `pubof` is.

4.2 The Axioms

Pairing Axiom The operations `frst` and `scnd` are the usual projections characterizing pairs.

$$\texttt{pair}\ r_1\ r_2 = r \ \leftrightarrow\ \texttt{frst}\, r \downarrow r_1 \wedge \texttt{scnd}\, r \downarrow r_2 \tag{1}$$

Axiom about `gen` The `gen` operation delivers values of the appropriate sorts.

$$\texttt{rtsort}(\texttt{gen}\ n\ srt) = srt \tag{2}$$

Axioms about `pubof` The `pubof` partial function makes a bijection from sort **ikey** to the sort **akey** (and is undefined off of the sort **ikey**).

$$\texttt{pubof}\, r_1 = r_2 \rightarrow \text{sort}(r_1) = \textbf{ikey} \wedge \text{sort}(r_2) = \textbf{akey} \tag{3}$$

$$\text{sort}(r_1) = \textbf{ikey} \rightarrow \exists!\, r_2, \texttt{pubof}\, r_1 = r_2 \tag{4}$$

$$\text{sort}(r_2) = \textbf{akey} \rightarrow \exists!\, r_1, \texttt{pubof}\, r_1 = r_2 \tag{5}$$

Encryption Axiom The standard relationship between encryption and decryption is of course

$$\texttt{encr}\ r_p\ r_{ke}\ r_e\ \leftrightarrow \texttt{decr}\ r_e\ (r_{ke})^{-1} = r_p$$

when expressed using the runtime inverse function $(-)^{-1}$. Since the inverse function is not feasibly computable we prefer the following formulation in terms of `rtinv`.

$$\texttt{encr}\ r_p\ r_{ke}\ r_e \wedge (\texttt{rtinv}\ r_{ke}\ r_{kd}) \rightarrow \texttt{decr}\ r_e\ r_{kd} = r_p \tag{6}$$

$$\texttt{decr}\ r_e\ r_{kd} = r_p \wedge (\texttt{rtinv}\ r_{ke}\ r_{kd}) \rightarrow \texttt{encr}\ r_p\ r_{ke}\ r_e \tag{7}$$

Summary: the Runtime Theory

Definition 4.4. The theory $\mathcal{R}$ comprises the axioms (1), (2), (3), (4), (5), (6), and (7) .

In the Coq code for MOLLY we use a typeclass to capture the theory $\mathcal{R}$.

5 Compilation

Here we outline the structure of the compilation process.

The notion of "saturation" of a proc plays a key role, and we present a careful discussion of saturation Section 6. But for an intuition, imagine generating code to process a reception of a term (Pr (Dt 1) (Dt 2)). We will bind (Pr (Dt 1) (Dt 2)) to a new location v, but will also want to generate code that serves to ensure that an incoming value at runtime is of the right form. We do this bind emitting code for a bonding of t_1 to another location v_1 with the constraint that v_1 is equal to (Frst v). That (plus the corresponding process for t_2) serves to check that and incoming value really is a pair, since otherwise the (Frst v) (or the (Scnd v) will fail at runtime). Then we need code to ensure that the value corresponding to t_1 really is of type **data** ... and so forth.

Saturation is the process of emitting all the bindings and checks required for a proc to be closed under these obligations.

5.1 The Main Loop

The overall structure of the compiler is simple. Given an input role rl, we

1. initialize a proc pr with generated values, as explained below
2. then loop through the actions of the role:
   (a) if the current action is an Prm or a reception Rcv of a term t we add an Event statement to pr recording the input or reception; then add a statement binding t to a new location; then saturate pr
   (b) if the current action is an Ret or a Snd of a term t we add an Event statement to pr recording the output or transmission; then saturate pr
3. since saturation can fail, the return type of each of the above steps returns, and of the compilation as a whole, is $\text{proc}_\perp$.

The state of the compilation at any point is a record with the following data

- the role to be compiled
- the current proc
- a list *done* of the role actions treated so far
- a list *todo* of the role actions yet to be treated

5.1.1 Invariants

To express our invariants we first note that the Bind statements of any proc naturally build a relation β from terms to locations:

Definition 5.1. The relation β from terms to locations is defined as

$$(\beta\ t\ v) \text{ if for some } e, (\mathsf{Bind}\ (t, l)\ e) \text{ is in } \mathsf{pr}.$$

Using β we define the following invariants maintained by the compilation process.

1. The concatenation of *done* with *todo* is the original role
2. The relation β systematically relates the list of terms *done* and the trace of pr. More precisely we have

$$\mathrm{map}_{\mathrm{R}}\ \beta^{\mathsf{Act}}\ \mathit{done}\ \ \mathsf{pr}$$

3. pr is saturated.

These invariants lead to the following properties of a successful compilation returning a proc pr.

- From invariants 1 and 2: the role rl and the trace of pr are related as

$$\mathrm{map}_{\mathrm{R}}\ \beta^{\mathsf{Act}}\ \ \mathsf{rl}\ \ \mathrm{trace}\,\mathsf{pr}$$

- pr is saturated

Those closure properties are key for ensuring our core correctness property, the Reflecting Transcripts property. In Section 6 we explain those properties and the mechanisms to ensure them; in Section 9 we present the proof of the Reflecting Transcripts theorem.

5.2 Initialization

Note the difference in character between the term (Dt 3) and the terms (Dt 1) and (Dt 2) in the role below.

```
...
(Rcv (Ch 1) (En (Dt 1) (Hs (Dt 2))));
...
(Snd (Ch 1) (Dt 3))
```

...

Assuming there are no occurrences of (Dt 3) prior to the one shown, the term (Dt 3) is to be freely chosen by the agent executing the role.

We assume that the runtime has a mechanism for constructing values of a given sort. Correspondingly our procs have an expression Genr intended to be implemented by this mechanism.

We next explain how the compiler emits the necessary statements. Key pairs require some special treatment, which we explain after giving the routine case.

5.2.1 Generation Bindings

Definition 5.2 (Polarity)**.** Let rl be a role and let t be an elementary term occurring as a subterm in rl.

We say that has *negative polarity* in rl if the first event of rl in which t is a subterm is either a Param or a Rcv.

We say that has *positive polarity* in rl if the first event of rl in which t is a subterm is either a Ret or a Snd.

For each elementary term t with positive polarity which is not an asymmetric key, the compiler emits code binding t to a location with a corresponding Genr expression. Specifically, we emit the statement

$$\mathsf{Bind}\ (t, v)\ (\mathsf{Genr}\ s\ k) \tag{8}$$

where v is a fresh location, s is the sort of t, and k is an natural-number index that identifies which occurrence of t is being treated.

There is a subtlety, though, about key pairs for asymmetric encryption, which we explain next.

Generation and Key Pairs Recall the definition of symbolic key pair from 3.3. It will be convenient to use the shorthand $(pri\ n)$ and $(pub\ n)$ for (Ik (Av n)) and (Ak (Av n)), respectively.

If both $(pri\ n)$ and $(pub\ n)$ have positive polarity, we should not generate calls to Genr independently for the two terms; this would lose the constraint that their values should be key pairs. And if exactly one of them has positive polarity then any occurrence of its key partner is to be received later. The proc will need code to check that this reception is suitable, that is, its value must make a runtime key pair with the value of the positive-polarity original term. This suggests that the symbolic key *partner* of the term with positive polarity must be associated with a binding.

The following process implements these ideas.

If either $(pri\ n)$ *or* $(pub\ n)$ *has positive polarity:* add the two bindings

$$\mathsf{Bind}\ ((pri\ n), v_{pri})\ (\mathsf{Genr}\ \mathbf{ikey}\ k) \tag{9}$$

$$\mathsf{Bind}\ ((pub\ n), v_{pub})\ (\mathsf{PubOf}\ v_{pri}) \tag{10}$$

where v_{pri} and v_{pub} are fresh locations and k is an index.

If neither $(pri\ n)$ nor $(pub\ n)$ has positive polarity, nothing happens for these terms during initialization: saturation will ensure that pr have appropriate CKypr checks to ensure that locations associated with these terms make a key pair.

Now, the semantics of procs will ensure that the values associated with v_{pri} and v_{pub} will be runtime key pairs, because of the $(\mathsf{PubOf}\ v_{pri})$ expression. But we should also check that when we generate a key pair then our proc has other runtime checks that we expect. For instance let us suppose first that $(pri\ n)$ occurs with positive polarity and that $(pub\ n)$ occurs with negative polarity, say by a binding

$$\mathsf{Bind}\ ((pub\ n), v_1)\ e.$$

We will want to know that pr has sufficient checks to ensure that the runtime values associated with v_{pri} and v_1 will make a key pair.

By our initialization process pr also has

$$\mathsf{Bind}\ ((pub\ n), v_{pub})\ (\mathsf{PubOf}\ v_{pri}) \tag{11}$$

But as a regular part of our proc construction we ensure that whenever two locations are assigned to the same elementary term we emit a assertions implying that the two locations have the same value at runtime. This is the Check Key Pair Condition of Definition 6.4. Specifically, in our current scenario we will emit assertions implying that v_{pub} and v_1 will have the same runtime values. This, in concert with the binding 11, ensures that the values associated with v_{pri} and v_1 will be runtime key pairs.

A similar argument shows that if $(pub\ n)$ occurs with positive polarity and that $(pri\ n)$ occurs with negative polarity, then our proc will have checks sufficient to ensure that any runtime values assigned to the corresponding variables will be runtime key pairs.

5.2.2 Generation and Initialization

The current compiler emits all code for generating values at the initialization phase. There is no necessity for doing it eagerly in this way: one could recognize generating terms "on the fly" when processing a send or return and do generation then. But it is convenient to do these Generations in the initialization phase since it makes analysis and proof of correctness a bit smoother.

5.3 The Structure of Expressions

Our proc expressions are not a recursive type per se: expressions are not arguments to expressions. But the binding structure in a proc gives an implicit recursive structure on expressions. For example, if e is the expression (Pair v_1 v_2) and our proc has bindings Bind (t_1, v_1) e_1 and Bind (t_2, v_2) e_2 then it is natural to think of e as being a pairing of e_1 with e_2.

Note there are exactly four expression operators that do not take locations as arguments:

Param, Read, Quot and Genr

and these take only indices serving to distinguish occurrences or to name a sort or a string.

So every expression built by the compiler can be thought of (modulo the indirection pointed out above involving locations) as a tree built by the other operators, whose leaves are essentially expressions built from simple Param, Read, Quot, and Genr expressions.

In Section 7.1 we show how this leads to a useful perspective on the overall structure of a proc.

6 Saturation

In the previous section we described saturation informally; here we define it carefully. First we introduce a convenient relation on locations. The proc statement (CSame v_1 v_2) compares the values in 2 given locations. We will need to work with the equivalence relation on locations this generates.

Definition 6.1 (Sameness)**.** The relation $\approx_{sm}$ is the least equivalence relation on locations such that $v_1 \approx_{sm} v_2$ whenever the statement (CSame v_1 v_2) is in pr.

6.1 Saturated Procs

We define two desirable conditions for a proc: that it be closed and justified. To be saturated (Definition 6.6) is to satisfy each of these.

Notation 6.2. To avoid verbosity we will use, in Definitions 6.4 and 6.5, the convention that writing

Bind (t, v) e

is shorthand for

Bind (t, v) e is one of the statements in the procedure pr.

6.1.1 Closure

Closure is the process of (i) adding binding statements to a proc in order to reflect the information known about parameters and messages received or generated and (ii) adding checks to reflect the constraints—such as sameness—among locations.

We first define the condition of "being closed" and then, in Section 6.2, give an algorithm for achieving closure.

6.1.2 Motivation for the Closure Conditions

- Pair Elimination: a proc satisfying this axiom is guaranteed to have statements in deconstructing a received value and storing the values derived for future use.
- Decryption: as for Pair Elimination, except that this axiom only ensures that we deconstruct a received encryption when the appropriate decryption key is available.
- Pair, Encryption, and Hash Introduction: these axioms ensure that we have statements deriving new values to be used for transmission and for potential use in constructing decryption keys or hash-values when received hashes are to be checked.
- Check Hash: since there is no operator to deconstruct a hash, we can only check the suitability of a value received when a hash is expected by comparing it to a known hash of that value.
- Check Same: ensures that we have code for the assertion that different locations bound to the same elementary term do indeed hold the same value.
- Check Sort: similarly to Check Same we need code for the assertion that a value received has the expected sort.
- Check Key Pair: is self-explanatory
- Check Quote: is self-explanatory

We need the following taxonomy on expressions for the definition and analysis of the closure conditions. The intuition for these definitions is that they capture circumstances where we do not need to do projections (out of pairs) or decryptions (of encryptions) during saturation.

Definition 6.3. Let pr be a proc and e and expression. Then e is

- a *pair expression for* term (Pr t_1 t_2) in proc pr if e is of the form (Pair l_1 l_2) and there are bindings in pr of the form Bind (t_1, l_1) $e_1$and Bind (t_2, l_2) e_2.

- an *encryption expression for* term $(\mathsf{En}\ t_1\ t_2)$ in proc pr if e is of the form $(\mathsf{Encr}\ l_1\ l_2)$ and there are bindings in pr of the form $\mathsf{Bind}\ (t_1, l_1)\ e_1$and $\mathsf{Bind}\ (t_2, l_2)\ e_2$.

Definition 6.4 (Closed Proc)**.** Let unv be a set of terms. A proc pr is *closed* if it satisfies the universal closures of the following formulas..

Pair Introduction Condition

$$\mathsf{Bind}\ (t_1, v_1)\ e_1 \wedge\ \mathsf{Bind}\ (t_2, v_2)\ e_2 \wedge\ (\mathsf{Pr}\ t_1\ t_2) \in \mathsf{unv} \rightarrow \exists v, \mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ (\mathsf{Pair}\ v_1\ v_2)$$

Encryption Introduction Condition

$$\mathsf{Bind}\ (t_1, v_1)\ e_1 \wedge\ \mathsf{Bind}\ (t_2, v_2)\ e_2 \wedge\ (\mathsf{En}\ t_1\ t_2) \in \mathsf{unv} \rightarrow \exists v, \mathsf{Bind}\ ((\mathsf{En}\ t_1\ t_2), v)\ (\mathsf{Encr}\ v_1\ v_2)$$

Hash Introduction Condition

$$\mathsf{Bind}\ (t_1, v_1)\ e_1(\mathsf{Hs}\ t_1) \in \mathsf{unv} \rightarrow \exists v, \mathsf{Bind}\ ((\mathsf{Hs}\ t_1), v)\ (\mathsf{Hash}\ v_1)$$

Pair Elimination Conditions

$$\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e \wedge\ e \text{ not a pair expression for } (\mathsf{Pr}\ t_1\ t_2) \rightarrow (\exists v_1, \mathsf{Bind}\ (t_1, v_1)\ (\mathsf{Frst}\ v) \wedge\ \exists v_2, \mathsf{Bind}\ (t_2, v_2)\ (\mathsf{Scnd}\ v)\ )$$

Decryption Condition

$$\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e \wedge\ e \text{ not an encryption expression for } (\mathsf{En}\ t_1\ t_2) \wedge\ \mathsf{Bind}\ (k^{-1}, v_1)\ e_1 \rightarrow \mathsf{Bind}\ (p, v_{\text{new}})\ (\mathsf{Decr}\ v\ v_1)$$

Check Hash Condition

$$\mathsf{Bind}\ ((\mathsf{Hs}\ t), v_h)\ e_h \wedge \mathsf{Bind}\ (t, v_t)\ e_t \rightarrow (\mathsf{CHash}\ v_h\ v_t)$$

Check Equality Condition

$$\text{elementary}\ t \wedge\ \mathsf{Bind}\ (t, v_1)\ e_1 \wedge \mathsf{Bind}\ (t, v_2)\ e_2 \rightarrow v_1 \approx_{sm} v_2$$

Check Quote Condition

$$\mathsf{Bind}\ ((\mathsf{Qt}\ s), v)\ e \rightarrow (\mathsf{CQot}\ v\ s)$$

Check Sort Condition

$$\text{elementary}\ t \wedge\ \mathsf{Bind}\ (t, v)\ e \rightarrow \exists v_1, v \approx_{sm} v_1 \wedge\ (\mathsf{CSrt}\ v\ (\text{sort}\, t))$$

Check Key Pair Condition

$$(t_1, t_2)\ \text{make a symbolic key pair} \wedge\ \mathsf{Bind}\ (t_1, v_1)\ e_1 \wedge\ \mathsf{Bind}\ (t_2, v_2)\ e_2 \rightarrow \exists v_1'\ v_2'\ e_1'\ e_2', \mathsf{Bind}\ (t_1, v_1')\ e_1' \wedge\ \mathsf{Bind}\ (t_2, v_2')\ e_2' \wedge\ v_1 \approx_{sm} v_1', \wedge\ v_2 \approx_{sm} v_2' \wedge\ (\mathsf{CKypr}\ v_1'\ v_2')$$

6.1.3 Being Justified

A proc pr is "justified" if, intuitively

- received encryptions always have decryption keys available, and
- whenever (Hs t) is bound in pr then t is also bound

Formally (we continue to employ Notation 6.2):

Definition 6.5 (Justified Proc)**.** A proc pr is *justified* if it satisfies

Encryption Justification

$$\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e \wedge\ \text{non-Encryption}\ e \rightarrow \exists v_1, \exists e_1, \mathsf{Bind}\ (k^{-1}, v_1)\ e_1$$

Hash Justification

$$\mathsf{Bind}\ ((\mathsf{Hs}\ t_1), v)\ e \wedge\ \text{non-Hash}\ \ e \rightarrow \exists v_1, \exists e_1, \mathsf{Bind}\ (t_1, v_1)\ e_1$$

Being justified is not a property that we can ensure of the procs the compiler builds. It is ultimately a property of the role we are compiling: it will fail if the parameters and expected receptions of the role do not provide the material needed to construct needed decryption keys or bodies of hashes.

6.1.4 Saturated

Definition 6.6 (Saturated Proc)**.** A proc pr is *saturated* with respect to a set of terms unv if it is closed with respect to unv and is justified.

We will be interested in the case where unv is the set of all subterms of terms occurring in a role to be compiled.

6.2 The Saturation Process

In this section we present an algorithm for taking a proc pr and returning a saturated extension of pr (or failing).

6.2.1 Motivation for the Closure Rules

The closure rules that follow are recipes for transformations that take an arbitrary proc pr (say, the state of a proc immediately after processing an event) and ultimately return a proc pr_* that satisfies the conditions of Definition 6.4.

The rules we present here do precisely this, in the sense that is a proc pr_* is a fixed point with respect to these rules then pr_* satisfies the the conditions of Definition 6.4. This is the content of Theorem 6.9.

For the most part the axioms of Definition 6.4 can be read "operationally" in the sense that they are Horn sentences which can be made true by augmenting the proc so that it satisfies the consequent whenever the antecedent is found to be true. But there are some small obstacles to interpreting the conditions of Definition 6.4 naively, specifically (i) witnessing the existential quantifiers in the consequents, and (ii) the logistics of ensuring that certain statements involving global relations such as $\approx_{sm}$ are satisfied.

The correspondence between the rules below and the axiomatic conditions Pair Elimination, Decryption, Pair Introduction, Encryption Introduction, and Hash Introduction and is clear: for a given instance of the antecedent of one of these conditions one need only (if necessary) generate a fresh location to witness the existential quantifier and generate appropriate Bind statements to establish the consequent. Thus several of the rules use the reference v_{new}: this is a reference to a variable not occurring elsewhere in the proc.

For the assertions Check Same, Check Sort, Check Hash, Check Quote, and Check Key Pair, the subtlety is that we want to ensure relationships between locations based on the sameness relation $\approx_{sm}$ without generating an excessive number of CSame statements. We do this by the trick of identifying and exploiting the "first location" for a term t: this will be the least location l such that a statement $\mathsf{Bind}\ (t, l)\ e$ occurs in pr. In fact there is nothing special about choosing the *least* location; all that matters is that there we identify one distinguished location for each elementary term bound in pr.

Once the rules have been identified, we saturate the proc arbitrarily by the rules. Section 6.2.3 explains why we cannot simply do a syntax-directed application of the inference rules.

6.2.2 The Closure Rules

The closure rules are defined in the context of a set unv of symbolic terms; in practice unv will be the set of subterms occurring in the role being compiled.

Definition 6.7 (Closure Rules)**.** Fix a universe unv of symbolic terms.

The eleven inference rules for *closing* a proc pr are the following.

Pair Introduction Rule

Here we require that $(\mathsf{Pr}\ t_1\ t_2)$ be in unv, and there are no bindings for $(\mathsf{Pr}\ t_1\ t_2)$ in pr.

$$\frac{\mathsf{Bind}\ (t_1, v_1)\ e_1 \qquad \mathsf{Bind}\ (t_2, v_2)\ e_2}{\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v_{\text{new}})\ (\mathsf{Pair}\ v_1\ v_2)}$$

Encryption Introduction Rule

Here we require that $(\mathsf{En}\ p\ k)$ be in unv, and there are no bindings for $(\mathsf{En}\ p\ k)$ in pr.

$$\frac{\mathsf{Bind}\ (p, v_1)\ e_1 \qquad \mathsf{Bind}\ (k, v_2)\ e_2}{\mathsf{Bind}\ ((\mathsf{Pr}\ p\ k), v_{\text{new}})\ (\mathsf{Pair}\ v_1\ v_2)}$$

Hash Introduction Rule

Here we require that $(\mathsf{Hs}\ t)$ be in unv, and there are no bindings for $(\mathsf{Hs}\ t)$ in pr.

$$\frac{\mathsf{Bind}\ (t, v)\ e}{\mathsf{Bind}\ ((\mathsf{Hs}\ t), v_{\text{new}})\ (\mathsf{Hash}\ v)}$$

Pair Elimination Left and Right Rules

We apply these rules when e not a pair expression for $(\mathsf{Pr}\ t_1\ t_2)$

$$\frac{\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e}{\mathsf{Bind}\ (t_1, v_{\text{new}})\ (\mathsf{Frst}\ v)} \qquad \frac{\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e}{\mathsf{Bind}\ (t_2, v_{\text{new}})\ (\mathsf{Scnd}\ v)}$$

Decryption Rule

We apply this rule when e not an encryption expression for $(\mathsf{En}\ p\ k)$ The active premise is the encryption binding.

$$\frac{\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e \quad \mathsf{Bind}\ (k^{-1}, v_1)\ e_1}{\mathsf{Bind}\ (p, v_{\text{new}})\ (\mathsf{Decr}\ \ v\ v_1)}$$

Check Hash Rule

The active premise is $\mathsf{Bind}\ ((\mathsf{Hs}\ t), v_h)\ e_h$

$$\frac{\mathsf{Bind}\ ((\mathsf{Hs}\ t), v_h)\ e_h \qquad \mathsf{Bind}\ (t, v_t)\ e_t}{(\mathsf{CHash}\ v_h\ v_t)}$$

Check Quote Rule

$$\frac{\mathsf{Bind}\ ((\mathsf{Qt}\ s), v)\ e}{(\mathsf{CQot}\ v\ s)}$$

Check Sort Rule

We apply this rule when t is an elementary term and v_1 is the first location for t in pr.

$$\frac{\mathsf{Bind}\ (t, v)\ e}{(\mathsf{CSrt}\ v\ \mathrm{sort}\, t)}$$

Check Same Rule

We apply this rule when t is an elementary term, $v_f < v_1$, and v_f is the first location for t in pr.

The active premise is the binding whose location is v_1.

$$\frac{\mathsf{Bind}\ (t, v_1)\ e_1 \qquad \mathsf{Bind}\ (t, v_f)\ e_f}{(\mathsf{CSame}\ v_1\ v_f)}$$

Check Key Pair Rule

We apply this rule when (t_1, t_2) makes a symbolic key pair, v_1 is the earliest location for t_1, and v_2 is the earliest location for t_2.

The active premise is the binding whose location is v_2.

$$\frac{\mathsf{Bind}\ (t_1, v_1)\ e_1 \qquad \mathsf{Bind}\ (t_2, v_2)\ e_2}{(\mathsf{CKypr}\ v_1\ v_2)}$$

6.2.3 Closure is not Syntax-Directed

As the examples below will demonstrate, when we use the rules to construct a saturated proc, we cannot apply them in a naively syntax-directed way. In the course of analyzing a reception we sometimes must use bindings that we can access but which have not yet themselves been fully analyzed.

Example 11. Suppose we receive the pair

$$((\mathsf{En}\ b\ k),\ (\mathsf{En}\ k\ (\mathsf{En}\ b\ k)))$$

For readability we temporarily revert to ordinary pair notation instead of writing

$$(\mathsf{Pr}\ \ (\mathsf{En}\ b\ k)\ (\mathsf{En}\ k\ (\mathsf{En}\ b\ k)))$$

Assuming b and k are elementary, we can successfully analyze this by (i) using the first component of the pair as decryption key for the second component, thereby obtaining k, then (ii) using k to decrypt the first component. The net result is that will generate bindings for each of

$$\{\ ((\mathsf{En}\ b\ k),\ (\mathsf{En}\ k\ (\mathsf{En}\ b\ k))), ((\mathsf{En}\ b\ k)),\ (\mathsf{En}\ k\ (\mathsf{En}\ b\ k)),\ k,\ b\ \}$$

///

We can also vary the example so that it does not depend on randomized operations being used as keys.

Example 12. Suppose we receive the pair

$$((\mathsf{Hs}\ (b,k)),\ (\mathsf{En}\ (b,k)\ (\mathsf{Hs}\ (b,k))))$$

Assuming b and k are elementary, we can successfully analyze this by strategy similar to the previous one. ///

The next example shows we may need to apply construction (i.e. introduction) rules in the course generating code for a reception.

Example 13. Suppose we receive the pair

$$((\mathsf{Hs}\ (b,k)),\ (\mathsf{En}\ (b,k)\ ((\mathsf{Hs}\ (b,k)), (\mathsf{Hs}\ (b,k)))))$$

Assuming b and k are elementary, we can successfully analyze this by using the first component to construct the pair $((\mathsf{Hs}\ (b,k)), (\mathsf{Hs}\ (b,k)))$, so that it can be used as a decryption key for $(\mathsf{En}\ (b,k)\ ((\mathsf{Hs}\ (b,k)), (\mathsf{Hs}\ (b,k))))$. ///

So our code proceeds by (after somewhat arbitrarily ordering the rules) applying the first rule that can fire and continuing until reaching a fixed point. We argue termination in Section 6.2.4.

6.2.4 Termination

Theorem 6.8. *Let* pr *be a* proc *and* unv *a set of terms. There are no infinite sequences of saturation rules starting with* pr *using* unv*.*

Proof. The three introduction rules apply at most once for each $t \in \mathsf{unv}$, and the new bindings they add cannot be premises of any other rule. Thus there are at most $|\,\mathsf{unv}\,|$ applications of introduction rules in any saturation process.

So it suffices to argue that there can be only finitely many application of Checks and the elimination rules **Pair Elimination Left and Right and Decryption**.

Let us say that a binding $\mathsf{Bind}\ (t, v)\ e$ is a *redex* if it is the active premise of a rule whose conclusion is not in pr.

Note that each binding can be a redex for at most one rule, with two exceptions: $(\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e)$ can be a redex for both Pair Elimination Left and Pair Elimination Right, and a binding for an asymmetric key can be a premise for Check Key Pair as well as for (one of) Check Sort or Check Same.

Let us assign a *weight* to each binding $(\mathsf{Bind}\ (t, v)\ e)$ in pr, by (i) counting the number of rules for which it is an active redex and (ii) multiplying this number by the size of t.

For example, if $(\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e)$ is in pr and neither the conclusion of Pair Elimination Left nor Pair Elimination Right is in pr then this binding gets weight $2|(\mathsf{Pr}\ t_1\ t_2)|$.

Then we say that the *weight* of pr is the sum of the weights of the bindings in pr. We claim that each elimination or Check rule application decreases this weight.

First: by inspection we see that when a rule fires, the active premise is no longer a premise for that rule.

Second: when a Check rule fires, the weight of pr decreases by the size of term being bound. No bindings are added by a Check rule.

Finally, when Pair Elimination Left or Pair Elimination Right or Decryption fires, the size of the term being bound is subtracted from the weight of pr, and replaced by the weight of some term in a new binding. But this new term is smaller than the term in the redex.

Thus the weight of the proc decreases at each step, and saturation must terminate. ///

6.2.5 Correctness

The rules in Definition 6.7 that create new bindings (for example, Pair Elimination) have an obvious relationship with the corresponding declarative conditions

in Definition 6.4. But the rules and conditions about check statements are more subtly linked, essentially because the rules only add individual statements to the proc yet the condition on the check statements make reference to more global properties of a proc, in particular, the $\approx_{sm}$ relation. The next theorem verifies that the rules are sufficient to enforce the conditions we need.

Theorem 6.9. *Fix a set of terms* unv. *Suppose* pr *is closed under the rules of Definition 6.7 (with* unv *as bounding set). Then* pr *is closed (with respect to* unv*) in the sense of Definition 6.4.*

Proof.

- *Verifying the* **Pair Introduction Condition**

  If Bind (t_1, v_1) e_1 and Bind (t_2, v_2) e_2 are in pr and (Pr t_1 t_2) is in unv then the Pair Introduction Rule ensures that there exists v with Bind ((Pr t_1 t_2), v) (Pair v_1 v_2) in pr.

- *Verifying the* **Encryption Introduction Condition**

  Just as for Pair Introduction, the Encryption Introduction Rule ensures this property directly.

- *Verifying the Verifying the* **Hash Introduction Condition**

  The Hash Introduction Rule ensures this property directly.

- *Verifying the* **Pair Elimination Condition**

  The Pair Eliminations Rules ensure this property directly.

- *Verifying the* **Decryption Condition**

  The Decryption Rule ensures this property directly.

- *Verifying the* **Check Hash Condition**

  The Check Hash Rule ensures this property directly.

- *Verifying the* **Check Quote Condition**

  The Check Quote Rule ensures this property directly.

- *Verifying the* **Check Equality Condition**

  Suppose t is elementary and Bind (t, v_1) e_1 and Bind (t, v_2) e_2 are in pr.

  Let v_0 be the first location for t in pr. Since pr is closed under the Check Same Rule, either

  - v_0 is v_1 and (CSame v_1 v_2) is in pr, or
  - v_0 is v_2 and (CSame v_2 v_1) in pr, or
  - v_0 is neither v_1 nor v_2 and both (CSame v_0 v_1) and (CSame v_0 v_2) are in pr

In all of these cases, $v_1 \approx_{sm} v_2$.

- *Verifying the* **Check Sort Condition**

  Suppose t is elementary and Bind (t, v) e is in pr.

  Take v_0 to be the first location for t in pr. Since pr is closed under the Check Sort Rule, (CSrt v_0 (sort t)) is in pr. By the just-proven fact that pr satisfies the Check Equality Condition, $v \approx_{sm} v_0$ as desired.

- *Verifying the* **Check Key Pair Condition**

  Suppose (t_1, t_2) makes a symbolic key pair and Bind (t_1, v_1) e_1 and Bind (t_2, v_2) e_2 are in pr.

  Take v_1' and v_2' to be the respective first-locations for t_1 and t_2. Then certainly $v_1 \approx_{sm} v_1'$ and $v_2 \approx_{sm} v_2'$. And (CKypr v_1' v_2') is in pr by the Check Key Pair Rule.

///

7 Some Results on Procs

7.1 The Structure of Expressions in Bindings

In Section 5.3 we observed that expressions in bindings can be thought of as the result of flattening of a recursive type using locations.

We noted that we can view expressions in the presence of proc bindings as a tree whose leaves are expressions built from simple Param, Read, Quot, and Genr expressions.

Now we can say more. The compiler makes initial bindings using Param and Read when translating an input or reception, respectively, and makes an initial bindings using uses PubOf and Genr when initializing or preparing a transmission, respectively. If we examine the subsequent saturation process we can see that

- the expressions built when processing a parameter or a reception are precisely those built
  - starting from Param and Read and
  - using the "destructive" operators

    Frst Scnd Decr
- the expressions built when initializing or processing a transmission or a return are precisely those built
  - starting from Genr, Quot, and PubOf and
  - using the "constructive" operators

    Pair Encr Hash

7.2 Procs and Derivability

In this section we show the intimate connection between the procs constructed by MOLLY and Dolev-Yao derivability of symbolic terms.

Roughly speaking, the connection is this: if pr is a proc generated from a role rl by Initialization and our Closure Rules, then the terms t such that there is a binding Bind (t, v) e in pr are the terms that are Dolev-Yao derivable from the input parameters and messages received in rl.

Of course the Genr operator allows us to bind *any* elementary term, so we have to exclude such generated terms. Also the Qt and PubOf forms are not treated in traditional Dolev-Yao so we need to enrich the system just slightly.

Definition 7.1 (Enhanced Dolev-Yao)**.** The *enhanced Dolev-Yao* inference system comprises the traditional Dolev-Yao system with the addition of the following two rules

1. derive (Qt s) for any string s
2. derive (Ak (Av n)) from (Ik (Av n))

Definition 7.2 (Obtained)**.** A term t is *obtained* in role rl if either (Prm t) is in rl or (Rcv ch t) is in rl for some ch

Before proving the next two lemmas we make a simple observation. The rules in Definition 6.7 that create a Bind are Pair Introduction, Encryption Introduction, Hash Introduction, Pair Elimination Left and Right, and Decryption. For each of these rules, if we suppress everything but the symbolic terms in the hypotheses and conclusion, we have an rule in the enhanced Dolev-Yao system.

Proposition 7.3. *Suppose* pr *is a* proc *generated from a role* rl *by our Initialization and Closure Rules.*

If Bind (t, v) e *is a statement in* pr *such that the expression* e *has no occurrence of the* Genr *operator, then* t *is derivable in the enhanced Dolev-Yao system from the set of terms obtained by* rl*.*

Proof. The proof is an easy induction on the construction of pr, using the observation immediately above. ///

Proposition 7.4. *Suppose* pr *is a* proc *generated from a role* rl *by our Initialization and Closure Rules. Further assume that* pr *is closed.*

If term t *occurs as a subterm of* rl *and is derivable in the enhanced Dolev-Yao system from the set of terms obtained by* rl*, then there exist* v *and* e *such that* Bind (t, v) e *is a statement in* pr*.*

Proof. Let $\mathcal{D}$ be a shortest derivation of t from the set of terms obtained by rl. The proof is by induction on $\mathcal{D}$, again using the fact that the enhanced Dolev-Yao rules are a kind of erasure of the Bind-creating closure rules. Since the closure rules have more structure than do Dolev-Yao inferences we need to check some details.

Suppose the last inference in $\mathcal{D}$ is a Dolev-Yao Pair Introduction, say of a term $(\mathsf{Pr}\ t_1\ t_2)$. By the induction hypothesis pr has bindings for t_1 and for t_2. The fact that $\mathcal{D}$ is a shortest derivation means that $(\mathsf{Pr}\ t_1\ t_2)$ has not already been derived, and so pr has no bindings for $(\mathsf{Pr}\ t_1\ t_2)$. That is, the side condition for the Pair Introduction closure rule is satisfied, and since pr is closed, the Pair Introduction closure rule will have fired, as desired.

The arguments for Encryption Introduction and Hash Introduction are just the same.

For Pair Elimination Left and Right and for Decryption the structure of the induction is similar but now as we apply the closure rules the side conditions are on the shape of the expressions e.

Consider the case of the Decryption Rule. We have, in $\mathcal{D}$, an inference

$$\frac{(\mathsf{En}\ p\ k) \quad k^{-1}}{p}$$

so we want to argue the pr has a binding for p. By induction we have, in pr, bindings of the form

$$\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e \quad \text{and} \quad \mathsf{Bind}\ (k^{-1}, v_1)\ e_1$$

Now we have two cases: either e is an encryption expression for $(\mathsf{En}\ p\ k)$ or not.

If so, then pr already contains a binding for p. If not, then we may fire the Decryption closure rule.

The argument for Pair Elimination is similar. ///

7.3 Executability

Several authors (*e.g.*, [CVB06],[BKRS15], [CR10]) have defined notions of *executability* of a protocol, statically-checkable properties that give confidence that a protocol can be run to completion.

We will eventually define *non-executability* as *failure of compilation*. To motivate that we start with the question: how can compilation fail?

Compilation succeeds precisely when saturation succeeds at each role event. Closure always halts: the process runs until no more rules can be applied, and our termination analysis say this will eventually halt (Theorem 6.8) . So the only thing that can go wrong is that we halt with a closed proc that isn't justified.

So suppose the proc pr is constructed from role rl and is closed, but not justified. This means (cf. Definition 6.5) that either

- there is a binding Bind $((\mathsf{En}\ p\ k), v)\ e$ in pr, with e not an Encr-expression, such that for no v_1, e_1 do we have Bind $(k^{-1}, v_1)\ e_1$ in pr, or
- there is a binding Bind $((\mathsf{Hs}\ t_1), v)\ e$ in pr, with e not a Hash-expression, such that for no v_1, e_1 do we have Bind $(t_1, v_1)\ e_1$ in pr.

Since in each case the terms are associated with a neutral expression, the proc needs to do a decryption or check-hash respectively.

But in the first case, the term $(\mathsf{En}\ p\ k)$ is derivable from the terms obtained in rl (Proposition 7.3), but the term k^{-1} is not derivable from the terms obtained in rl (Proposition 7.4).

Similarly, in the second case the term $(\mathsf{Hs}\ t_1)$ is derivable from the terms obtained in rl, but the term t_1 is not derivable from the terms obtained in rl

So failure of compilation is reflected by the existence of terms from the role that, by the results of the last section, cannot be Dolev-Yao derived but that are required in order for the proc to be able to construct statements it needs.

This analysis motivates the following definition.

Definition 7.5. A role rl is *non-executable* if the process of Initialization followed by closure under the rules of Definition 6.7 yields a proc which is not justified.

In this situation our compilation process does not return a runnable proc but in a precise way, it is not the compiler that is to blame: either there is an encryption derivable from the role whose decryption key is not derivable, or there is a derivable hash term whose body is not derivable. Each of these situations is one in which the proc at hand cannot successfully validate a necessary check.

Summarizing informally: we call a role *non-executable* if either some reception leads to an encryption whose decryption key cannot be derived, or some reception leads to a hash whose body cannot be derived. Non-executability can thus be determined statically: it is witnessed at compile-time by a failure of saturation.

Executable vs Non-Executable It is a bit awkward that we have been exploring *non-executability* as opposed to *executability*. But there is a good reason for that: to use the phrase *executable* as the negation of *non-executable* is quite a misleading choice of phrase!

To explain, let us unravel what it means for a proc pr to be **not** non-executable. This will mean that pr is closed and justified. But *execution of such a* pr *can still fail.*

This is a necessary consequence of the fact that encryption can be randomized. Different occurrences of t in the protocol may be associated with different locations in our proc, let us say $\mathsf{Bind}\ (t, v_1)\ e_1$ and $\mathsf{Bind}\ (t, v_2)\ e_2$. And if t is a term containing a randomized encryption as a subterm, then at runtime the locations v_1 and v_2 can have different values.

To see why this is a problem, return to the encryption case. Suppose pr has a binding $\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e$ with e neutral, so that the term $(\mathsf{En}\ p\ k)$ was derived from a reception. Since pr is justified we know that we have a binding for k^{-1}, say $\mathsf{Bind}\ (k^{-1}, v_1)\ e_1$. Saturation will have emitted a suitable decryption statement $\mathsf{Bind}\ (p, v_{new})\ (\mathsf{Decr}\ v\ v_1)$.

Now imagine that the encryption above is a symmetric encryption, so k^{-1} is k. But if k is a term itself containing a randomized encryption as a subterm, then it is possible at runtime that *the value of the occurrence of k used to build the encryption is not the same as the value of the occurrence of k used as decryption key.* This means that decryption will fail at runtime.

Similarly (more simply, in fact) suppose $(\mathsf{Hs}\ t_1)$ and t_1 are each bound, as required by being justified. A Check Hash test can fail if the two occurrences of t have different runtime values.

What's going on here is simply the fact that identity at the symbolic term level does not translate into identity of values as the bitstring level. This is not a weakness of the compiler. In a sense it is a lack of expressiveness of our symbolic term language, an unavoidable consequence of the fact that the randomness of encryption is not reflected in the syntax of terms.

The takeaway from this discussion is that although we can statically detect certain fatal obstacles to a role being able to be executed by compiled code, the absence of those obstacles does not guarantee that the compiled code will indeed run to completion. And this is why we hesitate to use the term "executable."

8 Valuations, Stores, and Transcripts

8.1 Raw Transcripts

Transcripts are sequences of runtime actions describing the observable behavior of a protocol execution.

Definition 8.1. A transcript is a list of (Act Rtval).

Example 14. The following is a transcript.

$$(\mathsf{Prm}\ r_1); (\mathsf{Rcv}\ r_1\ r_2); (\mathsf{Snd}\ r_1\ r_2)$$

This describes an execution in which

1. value r_1 is taken as a parameter;
2. a value r_2 is received at channel r_1;
3. that same value r_2 is sent over channel r_1. ///

Of course, it's not clear what this should mean if r_1 isn't in fact a channel value. And there will be more interesting structural constraints to be observed if we expect a transcript to be executions of roles or procs. In this section we define

1. what it means for a given transcript to denote a possible semantics of a given role rl (we will use the phrase "valid transcript for rl"), and
2. what it means for a given transcript to denote a possible execution of a given proc rl (we will use the phrase "valid transcript for pr").

We use the phrase "raw transcript" to refer to a list of (Act Rtval) without making any claims of validity.

8.2 Transcripts for a Role

To be a transcript for a role an (Act Rtval) sequence should satisfy two conditions.

First, the number of items and their character should match the role: for example, if the third element is the role is a Snd of a term then the the third element of the transcript should be a Snd of a runtime value, and so on.

Second, we expect a certain compositionality: for example if the role has (Pr t_1 t_2) as a reception and then t_1 as a transmission, and the transcript associates runtime value r with (Pr t_1 t_2) and associates runtime value r_1 with t_1 then r_1 should be the first projection of r.

The first constraint is straightforward to express. The second constraint needs some attention, and is the subject of Section 8.2.1, culminating in Definition 8.3.

Example 15. Consider this role

```
[(Prm (Ch 1)); (Prm (Ik ((Av 2))));
 (Rcv (Ch 1) (En (Nm 0) ((Ak ((Av 2))))));
 (Snd (Ch 1) ((Nm 0)))]
```

and consider the following raw transcript

```
(Prm r1); (Prm r2);
(Rcv r1 r3);
(Snd r1 r4)
```

By the way this is the first place where we see the technical utility of the Act parameterized data type: it is transparent how the actions over runtime values in the transcript are intended to correspond to the actions defined in the role.

This transcript describes an execution in which r_1 is a value assigned to channel 1; r_2 is a value assigned to (Ik (Av 2)) (the private key of the agent executing the role); r_3 is the value received on r_1; and r_4 is the value sent on r_1.

This transcript satisfies our first informally-stated constraint. But the more interesting second constraint requires (for example)

1. r_1 should be the sort of value that denotes a channel, r_2 should be a key, and so on.
2. r_3 should be a value that arises by encrypting r_4 with the key-partner of r_3

Being a "valid" transcript for a role will mean obeying sort constraints (as in (1) above) and relationship between values (as in (2) above). ///

Here is an important point: When a term such as (Ch 1) or (Nm 0) is used more then once in a role, the corresponding runtime value occurrences must be equal. That is, there should be a *function* from such terms to runtime values. But since we are working with randomized encryption, the value associated with a symbolic encryption (En p k) will not be uniquely determined by the values associated with p and k. Rather, there is a *relation* between symbolic terms that involve encryptions and runtime values.

This subtlety is reflected in Definition 8.2, where the fundamental construct that generates valid transcripts is a relation; this relation will be required to be functional on elementary terms.

8.2.1 Valuation for a Role

Let rl be a role and let tr be a transcript.

To say that the runtime events of tr are related to rl pointwise in a systematic way is to say that there is a relation τ from Term to Rtval such that

$$(\mathrm{map}_{\mathrm{R}}\, \tau^{\mathsf{Act}}\ \mathsf{rl}\, \mathsf{tr})$$

We may say that tr is *induced* by τ.

Next, in order to view tr as valid transcript for a role we will also insist on some compositionality conditions, articulated in the next definition.

Definition 8.2 (Valuation)**.** A relation $\tau \subseteq \mathrm{Term} \times \mathrm{Rtval}$ is a *term valuation* if it satisfies the following conditions.

1. τ is functional on elementary terms:
   For all t, r_1, r_2, if t is elementary and $(\tau\ t\ r_1)$ and $(\tau\ t\ r_2)$ then $r_1 = r_2$
2. τ respects sorts :
   For all t, r, if t is elementary and $(\tau\ t\ r)$ then $\mathrm{sort}\, t = \mathtt{rtsort}\ r$
3. τ respects pairing:
   For all t_1, t_2, r, if $(\tau\ (\mathsf{Pr}\ t_1\ t_2)\ r)$ then there exist r_1, r_2 such that $(\tau\ t_1\ r_1)$, $(\tau\ t_2\ r_2)$, and $\mathtt{pair}\, r_1 r_2 = r$.
4. τ respects hashing:
   For all t_1, r, if $(\tau\ (\mathsf{Hs}\, t_1)\ r)$ then there exist r_1 such that $(\tau\ t_1\ r_1)$ and $\mathtt{hash}\, r_1 = r$.
5. τ respects quote:
   For all s, r, if $(\tau\ (\mathsf{Qt}\ s)\ r)$ then $r = \mathtt{quot}\, s$
6. τ respects key pairs:
   For all t_1, t_2, r_1, r_2, if (t_1, t_2) makes a key pair, $(\tau\ t_1\ r_1)$, and $(\tau\ t_2\ r_2)$, then $\mathtt{kypr}\, r_1 r_2 = \mathit{true}$
7. τ has the disjunctive encryption condition:
   For all p, k_e, r_e, if $(\tau\ (\mathsf{En}\ p\ k_e)\ r_e)$ then at least one of the following holds:
   - (the encr condition) there exists r_p, r_{ke} such that
     - $(\tau\ p\ r_p)$
     - $(\tau\ k_e\ r_{ke})$
     - $\mathtt{encr}\, r_p r_k r_e$
   - (the decr condition) there exists r_p, r_{kd} such that
     - $(\tau\ p\ r_p)$
     - $(\tau\ (k_e)^{-1}\ r_{kd})$
     - $\mathtt{decr}\, r_e r_{kd} \downarrow r_p$

At first glance we might expect a condition requiring τ to respect inverse in the sense that if t_1 and t_2 are inverses, $(\tau\ t_1\ r_1)$, $(\tau\ t_2\ r_2)$, then $(\mathtt{rtinv}\ r_1\ r_2) = \mathit{true}$. But this is too much to ask given that τ is only a relation: if the sort of t is not **akey** or **ikey** then t can have several images under τ, which will certainly will not be runtime inverses of each other.

A valid transcript for a role rl is a raw transcript induced by a valuation.

Definition 8.3 (Valid Transcript for a Role)**.** Let rl be a role. A transcript tr is a *valid transcript for* rl if there exists a relation $\tau \subseteq \mathrm{Term} \times \mathrm{Rtval}$ such that

$$\tau \text{ is a valuation} \quad \text{and} \quad (\mathrm{map}_{\mathrm{R}}\, \tau^{\mathsf{Act}}\ \mathsf{rl}\, \mathsf{tr})$$

We will sometimes drop the word "valid," and sometimes say "tr is a transcript for pr" if no confusion can arise.

8.3 Transcripts for a Proc

As we did for roles, we want to define what it means for a raw transcript to be a suitable transcript for a given proc. This is completely straightforward.

The first observation is that procs have "internal" statements (bindings and checks) that will not contribute to transcripts. So we extract the statements we do care about:

Definition 8.4. Let pr be a proc. The body of pr is a sequence of statements. From this sequence we can extract the Events of the role and call this the *trace* of the proc.

The trace of a proc is a sequence of (Act Loc) statements. An execution of the proc is given by an assignment of runtime values to the locations of the proc.

Similarly as for roles we formalize the idea that a transcript tr "lines up" with the actions of proc pr by saying that there exists a relation σ : Loc $\to$ Rtval such that

$$(\mathrm{map_R}\ \sigma^{\mathsf{Act}}\ (\mathrm{trace}\ \ \mathsf{pr})\ \ \mathsf{tr})$$

But—just as for roles—we don't want to consider arbitrary associations of values to locations. To say that the transcript really is a "run" of the proc is simply to say that it arises as the proc executes its bindings and checks. So the relation between locations and runtime values will be simply the store of the proc as it executes.

Note the difference between the transcript notion for procs and for roles: for procs the relationship σ between locations and runtime values will be a *function* not just a relation. In a sense this reflects the fact that procs are deterministic programs.

So the correspondence condition induced by σ on the actions of pr and tr will simply be

$$\mathsf{tr} = \mathrm{map}\,\sigma^{\mathsf{Act}}(\mathrm{trace}\ \ \mathsf{pr})$$

Next we record what it means to be a store for a proc.

8.3.1 Store for a Proc

A store is a partial function from locations to runtime values that respects the intended semantics of the bindings and checks. A store σ is a "store for pr" if σ respects the statements of pr in the obvious way.

Definition 8.5 (Store)**.** Let pr be a proc and let σ : Loc $\to$ Rtval be a partial function on locations. Say that σ is a *store for* pr if σ respects the checks and the (expressions of the) bindings of pr in the following sense.

1. if pr has (CSrt v s) then `rtsort`$(\sigma v) = s$.

2. if pr has (CSame v_1 v_2) then $\sigma v_1 = \sigma v_2$
3. if pr has (CKypr v_1 v_2) then σv_1 and σv_2 make a runtime key pair, that is, `pubof` $(\sigma v_1) = (\sigma v_2)$
4. if pr has (CHash v_1 v_2) then $\sigma v_2 =$ `hash`(σv_1)
5. if pr has (CQot v s) then $\sigma v =$ `quot` s
6. if pr has Bind (t, v) (Pair v_1 v_2) then `pair`$(\sigma v_1)(\sigma v_2) = (\sigma v)$ holds
7. if pr has Bind (t, v_e) (Encr v_p v_k) then `encr`$(\sigma v_p)(\sigma v_k)(\sigma v_e)$ holds
8. if pr has Bind (t, v) (Hash v_1)
   then $(\sigma v) =$ `hash`(σv_1) holds.
9. if pr has Bind (t, v_1) (PubOf v)
   then `pubof`$(\sigma v) \downarrow (\sigma v_1)$ holds.
10. if pr has Bind (t, v_1) (Frst v)
    then `frst`$(\sigma v) \downarrow (\sigma v_1)$ holds
11. if pr has Bind (t, v_1) (Scnd v)
    then `scnd`$(\sigma v) \downarrow (\sigma v_1)$ holds
12. if pr has Bind (t, v_p) (Decr v_e v_k)
    then `decr`$(\sigma v_e)(\sigma v_k) \downarrow (\sigma v_p)$ holds.
13. if pr has Bind (t, v) (Quot s)
    then $(\sigma v) =$ `quot` s holds.

Definition 8.6 (Valid Transcript for a Proc)**.** Let pr be a proc. A transcript tr is a *valid transcript for* pr if there exists a partial function σ : Loc $\to$ Rtval such that

$$\sigma \text{ is a store for } \mathsf{pr} \text{ and } \mathsf{tr} = \operatorname{map} \sigma^{\mathsf{Act}} (\operatorname{trace} \mathsf{pr})$$

Example 16. Let pr be the following proc.

Proc 6: Proc for Example 16

```
1          Bind (Ch 1, L 1) (Param 1);
2          Csrt (L 1) Chan;
3          Evnt (Prm (L 2)); Bind (Nm 0, L 2) (Param
              2);
4          Csrt (L 2) Name;
5          Evnt (Prm (L 3));
6          Bind (Ik (Av 2), L 3) (Param 3);
7          Csrt (L 3) Ikey;
```

```
 8
 9          Evnt (Rcv (L 1) (L 4));
10          Bind (En (Nm 0) (Ak (Av 2)), L 4) (Read 1);
11          Bind (Nm 0, L 5) (Decr (L 4) (L 3));
12          Same (L 5) (L 2);
13
14          Evnt (Snd (L 1) (L 2))
```

Let the store σ be defined as follows.

$$\sigma v_1 = r_1 \qquad \sigma v_2 = r_0$$
$$\sigma v_3 = r_2 \qquad \sigma v_4 = r_3$$
$$\sigma v_5 = r_0$$

Then σ is a store for pr, as long as

- r_0 is a value of sort Nm, [line 4]
- r_1 is a channel value, [line 2]
- r_2 is a value of sort **ikey** [line 7]
- r_0 (the value at v_5) is a decryption of r_3 (the value at v_4) by r_2 (the value at v_3) [line 11 and line 12]

///

Example 17. Continuing Example 16, the following transcript tr

$$(\mathsf{Prm}\ r_1);\ (\mathsf{Prm}\ r_2); \tag{12}$$
$$(\mathsf{Rcv}\ r_1\ r_3); \tag{13}$$
$$(\mathsf{Snd}\ r_1\ r_4) \tag{14}$$

is a valid transcript for our proc.

To see why, first extract the trace of the proc:

Proc 7: Trace of the proc for Example 17

```
Evnt (Prm (L 2));
Evnt (Prm (L 3));
Evnt (Rcv (L 1) (L 4));
Evnt (Snd (L 1) (L 2))
```

It is easy to see that $\mathsf{tr} = \mathrm{map}\,\sigma^{\mathsf{Act}}(\mathrm{trace}\ \mathsf{pr})$ and we have seen in Example 16 that it arises from the valid transcript σ ///

9 Reflecting Transcripts

If role rl is compiled to proc pr, we want to know that any execution of pr is an execution of rl. This is captured by the *Reflecting Transcripts* theorem:

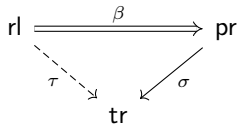
> **Theorem** Let rl be a role, and suppose that rl successfully compies to proc pr. Then any valid transcript for pr is a valid transcript for rl.

9.1 Outline of the Proof

Recall Definition 5.1, relating terms to locations based on the Bind statements in proc:

$$(\beta\ t\ v) \stackrel{\text{def}}{=} \quad \text{for some } e, (\mathsf{Bind}\ (t, l)\ e) \in \mathsf{pr}$$

Now suppose tr is a transcript for pr. By definition tr is determined by a store function, from Loc to Rtval. By precomposing this function with the relation β, we get a relation τ from Term to Rtval, as suggested by this picture.



This yields (modulo lifting these functions and relations to the Act data type) a raw transcript. It will not be hard to see that this transcript is in fact our original tr, and is induced by τ, and so tr is a *raw transcript for* rl, induced by τ. To establish that tr is a *valid transcript for* rl we need to show that it is induced by a valuation. This is where the Saturation conditions on pr come into play.

9.2 The Proof

Theorem 9.1 (Reflecting Transcripts)**.** *Let* rl *be a role, and suppose that* rl *successfully compies to* proc pr*. Then any transcript for* pr *is a transcript for* rl.

Proof. Let tr be a transcript for rl; let σ be a store for rl that induces tr. Viewing σ as a relation, let τ be the relational composition

$$\tau \stackrel{\text{def}}{=} \beta; \sigma$$

That is

$$\tau\ t\ r \stackrel{\text{def}}{=} \exists\ l\ e, \mathsf{Bind}\ (t, l)\ e \in \mathsf{pr} \wedge \sigma\ v \downarrow r$$

To establish that tr is a transcript for pr it suffices to show that τ is a valuation that induces tr.

First: to show that τ induces tr we want to show that

$$\mathrm{map_R}\ (\tau)^{\mathsf{Act}}\ \mathsf{rl}\ \ \mathsf{tr}$$

We have

$$\mathrm{map_R}\ (\beta^{\mathsf{Act}})\ \ \mathsf{rl}\ (\mathrm{trace}\ \ \mathsf{pr})$$

by definition of β.

We have

$$\mathrm{map_R}\ (\sigma^{\mathsf{Act}})\ (\mathrm{trace}\,\mathsf{pr})\ \ \mathsf{tr}$$

since tr is induced by σ.

By Lemma 3.6, then, we have

$$\mathrm{map_R}\ ((\sigma;\beta)^{\mathsf{Act}})\ \ \mathsf{rl}\ \ \mathsf{pr}, \text{ that is,}$$
$$\mathrm{map_R}\ (\tau^{\mathsf{Act}})\ \ \mathsf{rl}\ \ \mathsf{pr}$$

as desired.

Next we want to show that τ is a valuation.

We consider the clauses of Definition 8.2 in turn.

- τ is functional on elementary terms:

  Given t, r_1, r_2 with t elementary, $(\tau\ t\ r_1)$, and $(\tau\ t\ r_2)$. We want to show $r_1 = r_2$.

  By definition of $(\tau\ t\ r_1)$ and $(\tau\ t\ r_2)$ we have v_1, e_1, v_2, e_2 such that

  $$\mathsf{Bind}\ (t, v_1)\ e_1 \text{ and } \sigma(v_1) = r_1$$
  $$\mathsf{Bind}\ (t, v_2)\ e_2 \text{ and } \sigma(v_2) = r_2$$

  By the Check Equality Condition on pr we have $v_1 \approx_{sm} v_2$ in pr. Since σ respects the sameness checks of pr, we have $\sigma(v_1) = \sigma(v_2)$ as desired.

- τ respects sorts :

  Given t, r with t is elementary and $(\tau\ t\ r)$; we seek to establish that $\mathtt{rtsort}\ r = \mathrm{sort}\, t$.

  By definition of $(\tau\ t\ r)$ we have v and e such that

  $$\mathsf{Bind}\ (t, v)\ e \text{ is in } \mathsf{pr} \text{ and } \sigma(v) = r$$

  By the Check Sort Condition there is v_1 such that

  $$v \approx_{sm} v_1 \text{ and}$$

$$(\mathsf{CSrt}\ v_1\ (\mathrm{sort}\, t))\ \text{in}\ \mathsf{pr}.$$

Since σ respects CSrt,

$$\mathtt{rtsort}(\sigma v_1) = \mathrm{sort}\, t$$

and since $v \approx_{sm} v_1$,

$$\mathtt{rtsort}\, r = \mathtt{rtsort}(\sigma v) = \mathrm{sort}\, t$$

as desired.

- τ respects pairing:

  Given t_1, t_2, r with $(\tau\ (\mathsf{Pr}\ t_1\ t_2)\ r)$; we seek r_1, r_2 such that $(\tau\ t_1\ r_1)$, $(\tau\ t_2\ r_2)$, and $\mathtt{pair}\, r_1 r_2 = r$.

  By definition of τ we have v and e such that

  $$\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ e \text{ is in } \mathsf{pr} \text{ and } \sigma(v) = r.$$

  There are two cases: either e is a pair expression for $(\mathsf{Pr}\ t_1\ t_2)$ or not.

  1. If e is a pair expression for $(\mathsf{Pr}\ t_1\ t_2)$ then we know that e is of the form $(\mathsf{Pair}\ v_1\ v_2)$ and that there are v_1, v_2, e_1, and e_2 such that $\mathsf{Bind}\ (t_1, v_1)\ e_1$ and $\mathsf{Bind}\ (t_2, v_2)\ e_2$ are in pr.
     Set r_1 to be σv_1 and r_2 to be σv_2.
     Then
     $$(\tau\ t_1\ r_1) \quad \text{and} \quad (\tau\ t_2\ r_2).$$
     It remains to show $\mathtt{pair}\, r_1 r_2 = r$
     Since σ respects Pr and pr has
     $$\mathsf{Bind}\ ((\mathsf{Pr}\ t_1\ t_2), v)\ (\mathsf{Pair}\ v_1\ v_2)$$
     we have
     $$\mathtt{pair}\ (\sigma v_1)\ (\sigma v_2) = (\sigma v)$$
     which is to say
     $$\mathtt{pair}\ r_1\ r_2\ r$$
     as desired.
  2. If e is not a Pair expression for $(\mathsf{Pr}\ t_1\ t_2)$ then we use the fact that the runtime satisfies
     $$\mathtt{pair}\ r_1\ r_2 = r\ \leftrightarrow\ \mathtt{frst}\, r = r_1 \wedge \mathtt{scnd}\, r = r_2$$
     and so we exhibit r_1 and r_2 with $\mathtt{frst}\, r = r_1 \wedge \mathtt{scnd}\, r = r_2$.
     Since e is not a pair expression for $(\mathsf{Pr}\ t_1\ t_2)$, the Pair Elimination Conditions hold. So we have v_1 and v_2 such that
     $$\mathsf{Bind}\ (t_1, v_1)\ (\mathsf{Frst}\ v)$$
     $$\mathsf{Bind}\ (t_2, v_2)\ (\mathsf{Scnd}\ v)$$
     Set $r_1 = \sigma v_1$ and $r_2 = \sigma v_2$

Since σ respects Frst and Scnd,

$$\mathtt{frst}(\sigma v) \downarrow (\sigma v_1) \text{ and } \mathtt{scnd}(\sigma v) \downarrow (\sigma v_2)$$

that is,

$$\mathtt{frst}\, r \downarrow r_1 \text{ and } \mathtt{scnd}\, r \downarrow r_2$$

as desired.

- τ respects hashing:
  Given t, r with $(\tau\ (\mathsf{Hs}\ t)\ r)$; we seek r_t such that $(\tau\ t\ r_t)$ and $\mathtt{hash}\, r_t = r$.
  By definition of τ we have v_h, e_h, such that

  $$\mathsf{Bind}\ (t, v_h)\ e_h \text{ and } \sigma(v_h) = r$$

  By the Hash Justified Condition on pr there are v_t and e_t with

  $$\mathsf{Bind}\ (t, v_t)\ e_t \text{ in } \mathsf{pr}$$

  Then by the Check Hash Condition on pr

  $$(\mathsf{CHash}\ v_h\ v_t) \text{ is in } \mathsf{pr}$$

  Take r_t to be σr_t; since σ respects CHash,

  $$\begin{aligned} \sigma v_h &= \mathtt{hash}(\sigma v_t), \quad \text{that is,} \\ r_h &= \mathtt{hash}\, r_t \end{aligned}$$

- τ respects quote:
  Given s, r with $(\tau\ (\mathsf{Qt}\ s)\ r)$; we want to show $r = \mathtt{quot}\, s$
  By definition of τ we have v, e such that

  $$\mathsf{Bind}\ ((\mathsf{Qt}\ s), v)\ e \text{ is in } \mathsf{pr} \text{ and } \sigma(v) = r$$

  By the Check Quote Condition

  $$(\mathsf{CQot}\ v\ s) \text{ is in } \mathsf{pr}$$

  Since σ respects CQot, $\sigma v = \mathtt{quot}\, s$

- τ respects key pairs
  Given t_1, t_2, r_1, r_2, with t_1 and t_2 making a symbolic key pair and $(\tau\ t_1\ r_1)$, $(\tau\ t_2\ r_2)$; we want to show $\mathtt{kypr}\ r_1\ r_2 = true$.
  By definition of τ we have v_1, e_1, v_2, e_2 such that

  $$\mathsf{Bind}\ (t_1, v_1)\ e_1 \text{ is in } \mathsf{pr} \text{ and } \sigma(v_1) = r_1$$

$$\mathsf{Bind}\ (t_2, v_2)\ e_2 \text{ is in } \mathsf{pr} \text{ and } \sigma(v_2) = r_2$$

By the Check Key Pair Condition

$$\exists v_1'\ v_2'\ e_1'\ e_2', \mathsf{Bind}\ (t_1, v_1')\ e_1' \wedge\ \mathsf{Bind}\ (t_2, v_2')\ e_2' \wedge \\ v_1 \approx_{sm} v_1', \wedge\ v_2 \approx_{sm} v_2' \wedge\ (\mathsf{CKypr}\ v_1'\ v_2')$$

Let $r_1' = \sigma r_1$ and $r_2' = \sigma r_2$.

Since $v_1 \approx_{sm} v_1', \wedge\ v_2 \approx_{sm} v_2'$, and the fact that σ respects $\approx_{sm}$ it suffices to show that $\texttt{kypr}\ r_1'\ r_2' = true$.

But that follows from the facts that $(\mathsf{CKypr}\ v_1'\ v_2')$ is in pr and σ respects CKypr

- τ respects encryption:

  Given p, k_e, r_e, with $(\tau\ (\mathsf{En}\ p\ k_e)\ r_e)$; we want to establish the disjunctive encryption property.

  By definition of τ we have v and e such that

  $$\mathsf{Bind}\ ((\mathsf{En}\ p\ k_e), v)\ e \in \mathsf{pr} \tag{15}$$
  $$\sigma(v) = r_e. \tag{16}$$

  There are two cases: either e is an encryption expression for $(\mathsf{En}\ p\ k_e)$ or not.

  1. Suppose e is an encryption expression for $(\mathsf{En}\ p\ k_e)$, so that

     $$\mathsf{Bind}\ ((\mathsf{En}\ p\ k_e), v_e)\ (\mathsf{Encr}\ v_p\ v_k) \in \mathsf{pr} \tag{17}$$

     for some v_p and v_k. We establish the encr condition, *i.e.*, that there exists r_p, r_{ke} such that

     - $(\tau\ p\ r_p)$
     - $(\tau\ k_e\ r_{ke})$
     - $\texttt{encr}\ r_p\ r_k\ r_e$

     Since e is an encryption expression for $(\mathsf{En}\ p\ k_e)$, there are e_p and e_k such that

     $$\mathsf{Bind}\ (t_p, v_p)\ e_p \in \mathsf{pr} \tag{18}$$
     $$\mathsf{Bind}\ (t_k, v_k)\ e_k \in \mathsf{pr} \tag{19}$$

     Set r_p to be (σv_p) and r_{ke} to be (σv_k). Then

     $$(\tau\ p\ r_p) \tag{20}$$
     $$(\tau\ k_e\ r_{ke}). \tag{21}$$

Since σ respects Encr we have, by (17),

$$\mathtt{encr}(\sigma v_p)(\sigma v_k)(\sigma v)$$

which is to say

$$\mathtt{encr}\, r_p\ r_k\ r_e. \tag{22}$$

The encr condition follows from (20), (21), and (22)

2. Suppose e is not an encryption expression for $(\mathsf{En}\ p\ k_e)$. We establish the decr condition, *i.e.*, that there exist r_p and r_{kd} such that
   - $(\tau\ p\ r_p)$
   - $(\tau\ {k_e}^{-1}\ r_{kd})$
   - $\mathtt{decr}\ r_e\ r_{kd} \downarrow (\sigma v_p)$

   By the Encryption Justification property applied to (15) there are v_{kd} and e_{kd} such that

$$\mathsf{Bind}\ ((k_e)^{-1}, v_{kd})\ e_{kd} \in \mathsf{pr} \tag{23}$$

   Set r_{kd} to be (σv_{kd}), thus

$$\tau(k_e)^{-1} r_{kd} \tag{24}$$

   By the Decryption Condition applied to (15) and (16) we have

$$\mathsf{Bind}\ (p, v_p)\ (\mathsf{Decr}\ \ v\ v_{kd}) \in \mathsf{pr} \tag{25}$$

   for some v_p. Set r_p to be (σv_p), thus

$$\tau\ p\ r_p \tag{26}$$

   Since σ respects Decr, we have, by (25),

$$\mathtt{decr}\ (\sigma v)(\sigma v_{kd}) \downarrow (\sigma v_p)$$

   which is to say

$$\mathtt{decr}\ \ r_e\ r_{kd} \downarrow r_p \tag{27}$$

   The decryption condition follows from (24), (26), and (27)

///

Remark 9.2. It is instructive to compare the treatments of pairing and encryption in the above proof. We start with

$$\mathsf{Bind}\ ((\mathsf{Pr}\ \ t_1\ t_2), v)\ e \quad \text{or} \quad \mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e$$

In each instance the argument branched on whether or not the expression e was a Pair expression, or Encryption expression, respectively. In the affirmative case for each instance we argued directly that τ satisfied the definition of valuation, and the arguments were precisely parallel.

In the neutral cases we argued indirectly:

- using the axiom

$$\texttt{pair}\ r_1\ r_2 = r \ \leftrightarrow\ \texttt{frst}\, r = r_1 \wedge \texttt{scnd}\, r = r_2$$

  for pairing, and

- using the "decr condition" for encryption.

The pairing case was simpler. Why, for the encryption case, did we not just invoke the equivalence similar to that for pairing, namely

$$\texttt{encr}\ r_p\ r_k\ r_e \ \leftrightarrow\ \texttt{decr}\ r_e\ {r_k}^{-1} = r_p$$

instead of going to the trouble of defining the decr condition? Here's the explanation.

In the encryption proof, the runtime value r is the value of the store σ on the location for the key k. The equivalence about encryption refers to both r_k and ${r_k}^{-1}$. The latter would arise naturally as the value of σ on k^{-1}. But there is no reason to suppose that our proc pr has locations corresponding to each of k and k^{-1}. The situation for pairing is simpler in that the pairing equivalence involves no "alien" value analogous to ${r_k}^{-1}$. Logically speaking, deconstructing an encryption involves a minor premise, not so for deconstructing a pair.

In essence our proof in the encryption case is branching on whether the proc has a binding for k or a binding for k^{-1}; in the latter case we are using the fact that in a saturated proc an encryption binding with a neutral expression is *justified*.

This (natural!) inconvenience that pr probably does not have locations corresponding to each of k and its inverse is precisely why the definition of valuation has its disjunctive character.

Section 10.2 outlines an alternative approach to the semantics and our proofs that is related to this issue.

10 Discussion

10.1 Preserving Transcripts

It is natural to consider a converse to Reflecting Transcripts, namely that if role rl is compiled to proc pr, then any activity consistent with rl is a possible execution of pr. Formally, we might seek a result claiming:

> *If* rl *compiles to* pr *then any transcript for* rl *is a transcript for* pr.

But in fact a "preserving transcripts" in this form does not hold for Molly.

Example 18. Suppose ch is a channel term, k is a symmetric key, and p is arbitrary. Consider the artificially simple role containing two transmissions of the randomized encryption $(\mathsf{En}\ p\ k)$.

$$(\mathsf{Prm}\ ch); (\mathsf{Prm}\ p); (\mathsf{Prm}\ k)$$
$$(\mathsf{Snd}\ ch\ (\mathsf{En}\ p\ k))$$
$$(\mathsf{Snd}\ ch\ (\mathsf{En}\ p\ k))$$

Now consider a term valuation τ in which the term $(\mathsf{En}\ p\ k)$ is associated with a non-singleton set R of values. Then τ supports many transcripts in which the last two values sent are different values in R.

But any compiler that allocates only one location v for the term $(\mathsf{En}\ p\ k)$ cannot have a valid transcript with different values in its last 2 places. If σ is a store giving rise to a transcript for proc, the last two values must both be $\sigma(v)$. ///

This little example shows that any compiler supporting a Preserving Transcripts theorem as stated above must, at least, allocate as many locations to each role-term as there are occurrences of that term. This seems artificial, especially since the role specification language we have worked with does not provide any support for ascribing different behaviors with different occurrences of a given term.

10.2 Strong Term Valuations

This section should shed some light on the notion of valuation for a role, specifically the definition of a valuation "respecting encryption."

Here is another natural definition of valuation for a term. It replaces the disjunctive character of our official definition of respecting encryption, and strengthens the requirement concerning key pairs (compare Definition 8.2).

Definition 10.1 (Strong Valuation)**.** A relation $\tau \subseteq \mathrm{Term} \times \mathrm{Rtval}$ is a *strong term valuation* if it satisfies the conditions in Definition 8.2 with the following two strengthening of the conditions there about encryption and key pairs.

- τ strongly respects key pairs:

  For all t_1, t_2, r_1,

  - if (t_1, t_2) makes a key pair and $(\tau\ t_1\ r_1)$, then
    - there exists r_2 with $(\tau\ t_2\ r_2)$, and
    - for all r_2 with $(\tau\ t_2\ r_2)$, $\mathtt{kypr}\, r_1 r_2 = \mathit{true}$
  - if (t_2, t_1) makes a key pair and $(\tau\ t_2\ r_2)$, then
    - there exists r_1 with $(\tau\ t_1\ r_1)$, and
    - for all r_1 with $(\tau\ t_1\ r_1)$, $\mathtt{kypr}\, r_2 r_1 = \mathit{true}$

- τ strongly respects encryption:

  For all p, k_e, r_e, if $(\tau\ (\mathsf{En}\ p\ k_e)\ r_e)$ then there exists r_p, r_{ke} such that

  - $(\tau\ k_e\ r_{ke})$
  - $(\tau\ p\ r_p)$
  - $\mathtt{encr}\, r_p r_{ke} r_e$

Clearly if τ is a strong valuation then it is a valuation. What we show in this section is that if τ is a valuation that fails to be a strong valuation there is a natural way to extend it to a strong valuation.

Recall that when r is a runtime value we use r^{-1} to refer to the unique value r' acting as runtime inverse of r (Definition 3.9), even though r' might not be feasibly computable from r.

Definition 10.2. Let τ be a term valuation. Define the *completion* $\widehat{\tau}$ of τ to be

$$\tau\ \cup\ \{(t^{-1}, r^{-1}) \mid (t, r) \in \tau\}$$

Lemma 10.3. *If τ is a term valuation, then $\widehat{\tau}$ is a strong term valuation.*

Proof. Since the only difference between τ and $\widehat{\tau}$ is the addition of some pairs whose term is elementary, $\widehat{\tau}$ certainly respects pairing, hashing and quotes. It is easy to see that $\widehat{\tau}$ is functional on elementary terms and respects sorts. The strong key pair condition holds for $\widehat{\tau}$ by construction.

It remains to show that $\widehat{\tau}$ respects encryption in the sense of Definition 10.1.

So choose p, k_e, r_e such that $(\tau\ (\mathsf{En}\ p\ k_e)\ r_e)$. Since $\widehat{\tau}$ satisfies the decr condition we have r_p, r_{kd} such that

- $(\tau\ (k_e)^{-1}\ r_{kd})$
- $(\tau\ p\ r_p)$
- $\mathtt{decr}\, r_e\ r_{kd} = \mathit{Some}\ r_p$.

We seek r_p, r_{ke} such that

- $(\tau\ k_e\ r_{ke})$
- $(\tau\ p\ r_p)$
- $\mathtt{encr}\ r_p\ r_k\ r_e$.

Of course the r_p we seek is the given r_p; then take r_{ke} to be ${r_{kd}}^{-1}$. We have $(\tau\ (k_e)^{-1}\ r_{kd})$ by the construction of $\widehat{\tau}$, and $(\mathtt{encr}\ r_p\ r_k\ r_e)$ holds by the axiom

$$\mathtt{encr}\ r_p\ r_k\ r_e\ \leftrightarrow\ \mathtt{decr}\ r_e\ {r_k}^{-1} = r_p$$

///

Lemma 10.3 shows that we could have worked with strong valuations all along.

Strong valuation is perhaps more natural as a formalization of the meanings of symbolic terms in a role. But valuations as we originally defined them are the right definition for working with the data *explicitly available* from a given role: we typically only "have" one half of a key pair.

If we had used the notion of strong valuation when defining transcripts the main change in our development would come in the proof of the Reflecting Transcripts theorem. There we had to construct a valuation τ. This valuation was defined quite simply from the proc and the store as a composition. If we had needed to build a strong valuation we'd have built our (strong) valuation as the completion of this composition. It would have worked out but all the arguments would have been a bit more clumsy since we'd have had to take the completion into account at every step in the argument.

11 Future Work

Molly currently handles just a minimum set of primitives to exercise the relevant algorithm ideas and proof techniques. We expect that it will be straightforward to expand to other operations such as signatures and richer notions of tupling. Extensions to the message algebra that have interesting *semantic* consequences, such as Diffie-Hellman operations or rich equational theories, will demand some care but, we expect, no changes to our basic approach.

A more significant extension of the current work will connect it with protocol *analysis*. Specifically we plan to integrate Molly into the cpsa ecosystem, in a way such that a protocol designer, having established some security goals for her protocol using a cpsa analysis, can generate implementations of the various protocol roles, obtaining a version satisfying those goals. The main challenge here lies in the fact that our current correctness claims, based on transcripts, relate the symbolic and runtime traces of individual roles. To draw conclusion about the runtime semantics of a full protocol execution requires attention to the interactions among transcripts. This is a distributed activity involving a number of different compliant participants as well as possibly adversarial actions. We will turn to that larger question in a subsequent paper, using the current results as a basis.

Acknowledgments.

We are grateful to John Ramsdell for several helpful discussions.

References

[ABF17] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM (JACM)*, 65(1):1–41, 2017.

[AG97] Martín Abadi and Andrew D Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[AMV15] Omar Almousa, Sebastian Mödersheim, and Luca Viganò. Alice and Bob: reconciling formal models and implementation. *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, pages 66–85, 2015.

[AWL+22] Linard Arquint, Felix A Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N Wiesner, David Basin, and Peter Müller. Sound verification of security protocols: From design to interoperable implementations (extended version). *arXiv preprint arXiv:2212.04171*, 2022.

[AWL+23] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, and Peter Müller. Sound verification of security protocols: From design to interoperable implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, 2023.

[BBH12] Michael Backes, Alex Busenius, and Cătălin Hriţcu. On the development and formalization of an extensible code generator for real life security protocols. In *NASA Formal Methods Symposium*, pages 371–387. Springer, 2012.

[BKRS15] David Basin, Michel Keller, Saša Radomirović, and Ralf Sasse. Alice and Bob meet equational theories. *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday*, pages 160–180, 2015.

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Found. Trends Priv. Secur.*, 1(1-2):1–135, 2016.

[BN07] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.

[CJM98] Edmund Clarke, Somesh Jha, and Will Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proceedings, IFIP Working Conference on Programming Concepts and Methods (Procomet)*, 1998.

[CM05] Cas Cremers and Sjouke Mauw. Operational semantics of security protocols. In *Scenarios: Models, Transformations and Tools: International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, pages 66–89. Springer, 2005.

[CR10] Yannick Chevalier and Michaël Rusinowitch. Compiling and securing cryptographic protocols. *Information Processing Letters*, 110(3):116–122, 2010.

[CVB05] Carlos Caleiro, Luca Vigano, and David Basin. Deconstructing Alice and Bob. *Electronic Notes in Theoretical Computer Science*, 135(1):3–22, 2005.

[CVB06] Carlos Caleiro, Luca Vigano, and David Basin. On the semantics of Alice & Bob specifications of security protocols. *Theoretical Computer Science*, 367(1-2):88–122, 2006.

[DM00] Grit Denker and Jonathan Millen. Capsl integrated protocol environment. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 1, pages 207–221. IEEE, 2000.

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

[Gen69] G. Gentzen. Investigations into logical deduction (1935). In *The Collected Works of Gerhard Gentzen*. North Holland, 1969.

[JRV00] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 131–160. Springer, 2000.

[KB14] Michel Keller and Prof Dr David Basin. Converting Alice & Bob protocol specifications to Tamarin. *ETH Zurich*, 2014.

[LBH97] Gavin Lowe, Philippa Broadfoot, and Mei Lin Hui. Casper: a compiler for the analysis of security protocols. In *Protocols Proceedings of the 1997, IEEE. Computer society symposium on Research in security and Privasy*, pages 18–30, 1997.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[MK08] Jay McCarthy and Shriram Krishnamurthi. Cryptographic protocol explication and end-point projection. In *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*, pages 533–547. Springer, 2008.

[Möd09] Sebastian Mödersheim. Algebraic properties in Alice and Bob notation. In *2009 International Conference on Availability, Reliability and Security*, pages 433–440. IEEE, 2009.

[Mod14] Paolo Modesti. Efficient Java code generation of security protocols specified in AnB/AnBx. In *Security and Trust Management: 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings 10*, pages 204–208. Springer, 2014.

[Mod16] Paolo Modesti. Anbx: Automatic generation and verification of security protocols implementations. In *Foundations and Practice of Security: 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers 8*, pages 156–173. Springer, 2016.

[Pau97] Lawrence C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE CS Press, 1997.

[PGLSN22] Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[PJP15] Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*, pages 158–182. Springer, 2015.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study*. Almqvist and Wiksel, Stockholm, 1965.

[Ram21] John D Ramsdell. Cryptographic protocol analysis and compilation using CPSA and Roletran. In *Protocols, Strands, and Logic: Essays Dedicated to Joshua Guttman on the Occasion of his 66.66 th Birthday*, pages 355–369. Springer, 2021.

[SBCAP18] Riccardo Sisto, Piergiuseppe Bettassa Copet, Matteo Avalle, and Alfredo Pironti. Formally sound implementations of security protocols with JavaSPI. *Formal Aspects of Computing*, 30:279–317, 2018.

[TH05] Benjamin Tobler and Andrew CM Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Service*, pages 33–53. Springer, 2005.