Molly: A Verified Compiler for Cryptographic Roles

Daniel J. Dougherty
Worcester Polytechnic Institute
dd@wpi.edu

Joshua D. Guttman
The MITRE Corporation
guttman@mitre.org

***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. In order to define and prove the correctness of this compilation, we define *transcripts*, 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 may be randomized. Thus, at the runtime level we treat encryption as a relation rather than a function. Molly is developed in Coq, and includes a machine-checked proof that the procedure it constructs is correct with respect to the transcript semantics. Using Coq's extraction mechanism, one can build an efficient functional program for compilation.**

I. Introduction

Cryptoprotocols are short sequences of messages providing authentication and confidentiality, and establishing fresh shared keys. They are surprisingly hard to get right: active adversaries can manipulate compliant principals to obtain messages that undermine protocol goals. Moreover, they are surprisingly hard to implement correctly, as it may be ambiguous which checks to make on messages that are received.

A large body of research specifies protocols symbolically and uses formal methods to reason about their behavior. In this approach, different forms of protocol narrations describe cryptographic protocols. They are succinct and approachable but have varying semantics, and leave many implementation details implicit. Their semantics is complicated by the fact that the specifications of messages in protocol narrations generally consist of terms in a term algebra, while the actual messages in protocol executions are bitstrings. Moreover, since most cryptographic operations are randomized, the term-level specifications and the concrete messages are not connected by a function, but only by a relation.

When the protocol narration describes a message to be sent—generally specified as a term in a term algebra—a sequence of actions must construct a concrete instance of the specified transmission, and doing this correctly requires a degree of care. Received messages are even more difficult; we must orchestrate a sequence of checks to ensure the received concrete message is a genuine instance of the protocol narration description.

In this paper, we propose a solution to this challenge, namely generating the actions on concrete messages to construct outgoing messages and to check incoming messages. We also carefully present the correctness conditions that define this problem. We have proved that our solution satisfies these conditions using Coq, although this paper does not emphasize the specifics of the Coq formalization and proof.[1]

Generality is important, here. Many different cryptographic primitives might be used, and a compiler's correctness should be independent of the choice of (e.g.) one authenticated symmetric encryption or another. In our development, the assumption that matters most is that, when a successful decryption of ciphertext c with key k yields plaintext p, then c is among the many values that the randomized encryption of p with k could yield. This simple principle matches the notion of authenticated encryption well, and thus puts only the lightest constraint on what algorithm will be chosen to generate the encryptions. We will also assume a converse, namely that when c is a result of encrypting p with k, then decryption with k will succeed on c, yielding p. The corresponding principle for asymmetric encryption requires a little more care, as we must have a way to associate the two members k, k^{-1}, one of which will be used to encrypt, and the other of which may be used to decrypt. However, all of these requirements are light and easily formalized.

Pairing concrete messages together requires a similarly light constraint on pairing and the first and second projection operations π_1 and π_2, a sort of round-trip principle or unique parseability constraint on concrete messages:

> if $\pi_1(m) = p_1$ and $\pi_2(m) = p_2$,
> then m is the result of pairing p_1 and p_2.

Our framework remains correct for a wide variety of message formats and cryptographic operators, because it depends on very little about what the operations do. This motivates using an abstract protocol specification and compiling it to procedures that run on concrete messages. The rights and wrongs of the protocol and compilation are independent most of the choice of how to represent the messages and what crypto operators to use.

Which raises the question, what notion of *correctness* applies? In this paper, we consider only what applies to a compiler, generating procedures to execute locally on a single protocol participant. We do not need to do any protocol analysis or make cryptographic claims. Instead, we argue only that, whatever cryptographic primitives are appropriate to use for the protocol role definition, if those same primitives are linked against the compiler output, the resulting target code will execute only runs that the protocol definition permitted.

[1] A comprehensive account of this aspect of the work is on the arXiv [18] and the full Coq development is available on Github https://github.com/dandougherty/Molly.

Naturally, a good compiler will also generate target code that runs effectively in response to a large class of received messages; our compiler has this property also. However, we consider it important to separate the compiler correctness claim, which is a local claim that is parametric in the primitive operations, from global protocol analysis problems. We clarify our correctness claim in Section IV.

Abstract messages are terms in a term algebra; the concrete messages are bitstrings of some sort. Many ways of relating terms to bitstrings are possible, and we will refer to a particular scheme for representing terms via concrete messages as a protocol actually runs as a *runtime*. We impose only a few light constraints on runtimes: the axioms we assume are presented in Section VIII. The code generated by our compiler can be linked with one of a range of *runtime libraries* to format the messages as bitstrings in a particular way, including to implement cryptographic primitives. We often call concrete messages *runtime values*.

A. 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 protocol roles to *procs*; the latter are essentially the procs of Ramsdell's Roletran [35]. 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.

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. 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*.

A semantics for individual roles, as treated here, differs from a semantics for *protocol executions*, which comprise interactions between individual role executions. The former, our transcripts, are merely "sections" of the latter, consisting of the local events observable by one single principal.

Treatment of Randomized Encryption. Because many cryptographic operations are randomized, the bitstrings at runtime are not in one-to-one correlation with the symbolic terms of a role. 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 abstract terms to concrete runtime values (the "valuations" of Definition 1).

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 (Thm. 8), states that if role rl is compiled to procedure pr, any transcript for pr is a transcript for rl. The proof of the theorem is formally verified in Coq.

Proof-Theoretic View of Code Generation. We have organized our compilation according to the Dolev-Yao model [17], by which we mean that we view the principal executing a role as manipulating term-structured items according to derivation rules. Thus, reflecting much previous work, dating back at least to Paulson and Marrero et al. in the 1990s [30], [14], 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 [19], [34].

Compilability and Executability. This logical approach to compilation makes it easy to characterize the input roles on which compilation succeeds. 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.

B. Scope

Our goals in this paper are focused. We are not concerned with the details of cryptographic libraries, or of proofs that they meet their cryptographic requirements, on which much care has been expended by others. Indeed, our axiomatic approach to the operations on concrete messages means that verified cryptographic libraries fit directly with our approach; our work amplifies their value. Similarly, we have not concerned ourselves with the details of message formats, for instance the ASN.1 encodings; again, the axiomatic approach indicates light constraints for incorporating libraries of that kind. Moreover, we have not incorporated conditionals and other control structures into our source language. Indeed, our group has successfully added control structures and concrete message formats with considerable flexibility in related work on Zappa (mentioned in Section XI).

In this paper, we focus exclusively on the problem of relating actions on abstract terms with concrete messages, identifying a minimal core of ideas that suffice for expressing correctness and proving it in a fully mechanized way. This establishes the essential semantic core that justifies using particular implementation libraries or incorporating more flexible control.[2]

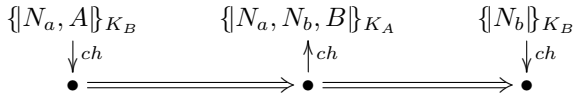
II. Example: The Needham-Schroeder-Lowe Responder

We consider a small example in order to introduce the main ideas of the compiler. We discuss it informally here and present

[2]The module Examples.v in the Coq development at https://github.com/dandougherty/Molly presents some concrete protocol roles and their translations.

it in full detail in the appendix.

The Needham-Schroeder-Lowe protocol is simple and familiar, which is helpful for our purposes. We will focus on the responder; our compiler works on different roles independently in any case. The procedure that implements the responder will need to be given a communications channel ch such as a socket to send and receive messages on. It also needs to be given its own name B as well as B's public and private keys as a *key pair* $(K_B, {K_B}^{-1})$. We will assume that it is also given the name A of an intended peer and A's public encryption key K_A. These are the *parameters* of the role. There are two receptions, the first and last events in the role, with a transmission in between them:



The bullets here represent the reception or transmission events, depending on the direction of the single arrow, and the double arrow represents the control flow: subsequent events cannot occur until previous events have succeeded. N_a is learnt by using K_B^{-1} to decrypt the encryption in the first message; N_b is to be generated freshly by B, who uses K_a to prepare the encryption; and N_b is confirmed in the last message after decryption using K_B^{-1}. A reception succeeds only when the incoming message is found to be of the required form, and a transmission succeeds only when the principal has constructed a message of the required form.

Thus, a protocol role specifies a sequence of actions, where the actions include transmissions and receptions. Actions also include receiving parameters at the start, as mentioned above. The value of N_a is learnt from the first message, and the value of N_b is generated by a library procedure.

We will define a protocol role specification to be a sequence of actions, where each action's content is a term in a term algebra, like those ones shown in the example above; of course Molly uses an ASCII rendering of the mathematical syntax above (Section VI).

Molly produces target code consisting of procedures affectionately known as procs. Procs are an intermediate language for representing straight-line, single assignment cryptographic programs. Procs can be translated——given a suitable cryptographic library——into a conventional imperative programs. A proc is a sequence of *statements* manipulating a set of *locations*. Each statement is an *Event*, a *Bind*, or a *Check*. Events express sends, receives, and input or output parameters: the sequence of events in a proc defines the *trace* of the proc. Checks are runtime checks, for example that a value stored in a location has the required sort, or that two locations have the same value, etc. A *Bind* statement Bind (t, v) e causes the value named by e to be stored into the location v, which never occurs as the target of another Bind statement. The term t has no effect at runtime, but in the compiler proof serves to express the invariant connection between the value e and an term in the input role specification; see Section VII, Eq. (1).

Thus, the output of Molly for the last reception of the NSL responder declares a receive event, and the message m received from ch is read into a location using a bind statement. An ensuing bind statement stores the plaintext resulting from a decryption of m into a location; the decryption key K_B^{-1} has already been bound to a location after being obtained as a parameter. The resulting value must equal the nonce B sent in the previous event; a *Check* statement ensures this. Finally, the results are returned to the caller.

The detailed version of this example in Section A sheds substantial light on how these mechanisms work.

III. Transcripts: Actions on Runtime Values

The externally visible actions of a protocol include accepting a parameter, returning a result, sending a message over a channel, and receiving a message over a channel. When the messages and parameter values all belong to a domain of runtime values, we call a sequence of these runtime actions a *transcript*.

Clearly, a transcript is incompatible with a role if, for example, the role starts by accepting a sequence of three parameters, while the transcript contains one runtime value as parameter followed by a message transmission, or if the role sends a message before receiving two, but the transcript receives two messages before sending one.

We say that a transcript tr is *compatible with* role rl, or rl-*compatible*, iff they are of the same length, and for each index up to the length tr engages in the same kind of action as rl.

This requires nothing about the runtime values of tr making sense relative to the algebraic terms contained in the successive actions of role rl. For instance, we want to insist that if role rl specifies receives a symbolic pair, then the corresponding runtime transcript value should be a runtime pair, and so on. To express that, we will associate runtime values to the subterms of terms in rl in a *valuation* such that the runtime operators relate runtime values as the terms are related (as in Def. 1).

Cryptographic operators being randomized, the valuation associating subterms and runtime values are relations rather than functions. The same term $t = \{|t_0|\}_K$ may be associated with different runtime values r, r' if they were generated at different moments with different random contribution. The relevant condition is that r and r' should each be possible results of encrypting some r_0 associated with t_0 with a runtime key associated with K.

A fine point is that a protocol participant may have a public encryption key K but not the corresponding decryption key K^{-1}, or conceivably K^{-1} but not K. Thus, when a role involves $\{|t_0|\}_K$, it may have used a runtime value associated with K to perform the encryption, or, if $\{|t_0|\}_K$ is part of a reception, it may have used one associated with K^{-1} to decrypt $\{|t_0|\}_K$ to obtain a runtime value associated with t_0.

Thus, the association may not provide runtime values for *all* of the syntactic subterms; for those K used only as keys it should provide either a runtime value associated with K or one associated with K^{-1}.

By contrast to terms containing encryptions, terms t for elementary values such as names, keys, or nonces should be associated with a single runtime value at all occurrences, since it is a protocol goal that these values should be fixed. This uniqueness is preserved by operations such as pairing and hashing which are functional, not randomized. Thus, when a pair term $t = (t_0, t_1)$ is associated with a runtime value r, there should be associations between t_0 and some r_0, and between t_0 and some r_1, such that $r = \texttt{pair}\, r_0 r_1$. Hashes induce a similar requirement. Thus:

Definition 1: Let RT be a model of the runtime theory $\mathcal{R}$. A relation $\tau \subseteq \mathrm{Term} \times \mathrm{RT}$ is a *valuation* iff:

1. $(\{|t_0|\}_K, r) \in \tau$ iff either (a) $\exists r_0, r_K$ s.t. $(t_0, r_0) \in \tau$, $(K, r_K) \in \tau$, and $\texttt{encr}\ r_0\, r_K\, r$; or else
   (b) $\exists r_0, r'_K$ s.t. $(t_0, r_0) \in \tau$, $(K, r'_K) \in \tau$, and $\texttt{decr}\ r\, r'_K\, r_0$;
2. $((t_0, t_1)), r) \in \tau$ iff $\exists r_0, r_1$ s.t. $(t_0, r_0) \in \tau$, $(t_1, r_1) \in \tau$, and $\texttt{pair}\ r_0\, r_1 = r$;
3. $(hash(t_0), r) \in \tau$ iff $\exists r_0$ s.t. $(t_0, r_0) \in \tau$ and $\texttt{hash}\ r_0 = r$;
4. τ is a function on elementary terms;
5. If K is an asymmetric private key, $(K, r) \in \tau$, and $(K^{-1}, r') \in \tau$, then $\texttt{pubof}\, r = r'$;
6. If t is elementary and $(t, r) \in \tau$, then $\mathrm{sort}\, t = \texttt{rtsort}\, r$;
7. $(\texttt{"}s\texttt{"}, r) \in \tau$ implies $r = \texttt{quot}\ s$.

A relation τ is an rl-*valuation* iff it is a valuation, and each term in an action of rl is in the domain of τ. ///

The first three clauses have a closure property on the domain as a consequence. We say that a set T of terms is *downward-closed* iff (i) $\{|t_0|\}_{t_1} \in T$ implies $t_0 \in T$ and either $t_1 \in T$ or $t_1^{-1} \in T$; (ii) $(t_0, t_1) \in T$ implies $t_0, t_1 \in T$; and (iii) $hash(t_0) \in T$ implies $t_0 \in T$.

Now a valuation τ has a downward-closed domain.

The clauses 1(a) and 1(b) establish the two cases of item (i) here, and clauses 2 and 3 establish (ii) and (iii).

Definition 2: Let rl be a role. Transcript tr is a rl-*transcript* iff it is rl-compatible and there exists an rl-valuation τ such that, for each i up to their common length, τ relates their i^{th} action. ///

Correspondingly, there is a notion of transcript for a proc pr. We naturally say that a transcript tr is pr-*compatible* if its actions match the actions in pr. Then, the analogue of a valuation in the proc context is that of a *store:* an assignment of runtime values to the locations of pr, and it is a pr-*store* iff the value associated with the location in the target of each Bind is a possible value of its source expression.

Definition 3: Let pr be a proc. Transcript tr is a pr-*transcript* iff it is pr-compatible and there exists a pr-store associating the location that is the target of each action to that runtime value in tr. ///

IV. Criterion of Correctness

The correctness condition for a compiler is that, if it compiles rl to pr, and tr is any pr-transcript, then tr is an rl-transcript also. We express this by saying that *transcripts are reflected*, i.e. back from the compilation target to its source.

One might expect to want a converse to this claim, which would state that every rl-transcript should be a possible pr-transcript, i.e. that no possible executions are lost. However, in the context of randomized cryptographic operations, this is unachievable. For instance, suppose a role is specified to receive a particular digital signature and retransmit it. The rl-transcripts as we defined them allow any sample of the randomized signature algorithm for the second, retransmitted occurrence, not just the same sample that was previously received.

However, a compiled proc pr receives a single sample, which it verifies and destructures. But depending on the signature algorithm, the recipient, lacking the signing key, may be unable to produce other samples. Thus, transcripts in which the runtime value retransmitted in the second action is a different sample are unimplementable for these signature algorithms.

Looking for a narrower definition of rl-transcripts would be arbitrary. In some cases, e.g. voting protocols, one wants an implementation to re-randomize cryptographic units, so as to unlink them for anonymity. It would be wrong to select an overall definition that would rule this out. Rather, one would want to adapt a compiler algorithm to emit procs to run in different regions of the space of runtime transcripts, possibly depending on the exact crytoprimitive in use.

Our correctness claim is a purely local, transcript-by-transcript claim. By contrast, a global protocol execution consists of a family of one or more transcripts representing the behaviors of different principals during a protocol run. We conjecture that these respect the predictions of symbolic protocol analysis.

V. Ideas Shaping the Compiler

At a high level, the process of compiling is as follows. Given an input role rl, we loop through the actions of rl, each of which is an input, output, transmission, or reception of a symbolic term. For each action we create an appropriate Bind statement relating (i) the symbolic term, (ii) a proc location, and (iii) a suitable proc expression. Furthermore, we then *saturate* the proc: here we introduce this notion informally here and describe it in more detail in Section VII-B.

Saturation: For an intuition about saturation, consider what we must do to generate code to process the reception of a value for the term (t_0, t_1). We will bind a reference to (t_0, t_1) 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 by emitting code for a binding of t_0 to another location v_0 together with the constraint that v_0 is equal to the first projection of v . This binding (plus the corresponding binding for t_1) serves to check that any incoming value really is a pair, since otherwise one or both of the projections will fail at runtime. We also need code to ensure that the value corresponding to t_1 really is of the right form . . . and so forth. When we get to the case of validating an elementary value such as a name or a key, we emit a suitable sort-check assertion.

Saturation is the process of emitting all the bindings and checks required for a proc to be closed under such obligations. It is defined formally as an inference system (see Section VII for sample inferences).

Saturation is not Syntax-Directed: There is a subtlety in the process of deconstructing a received value. 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 1: Suppose we receive the pair

$$(\{|b|\}_k, \{|k|\}_{(\mathsf{En}\ b\ k)})$$

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 we will generate bindings for the following four terms

$$\{|b|\}_k, \quad \{|k|\}_{\{|b|\}_k}, \quad k, \quad \text{and} \quad b$$

in addition to the original pair.

Saturation works by applying any rule that can fire and continuing until reaching a fixed point. This requires an termination argument (Theorem = 7).

Invariants: We maintain invariants for the compilation process that ensure that at the end of a successful compilation of role rl to proc pr,

1. The events of role rl and the trace of proc pr are related in the expected way, and
2. pr is saturated

These are the two properties that yield a proof of our core correctness result, the Reflecting Transcripts theorem.

VI. Data Structures

To describe the compilation process and its correctness precisely we require an introduction to the data structures used to represent roles and procs. It is convenient to first introduce a polymorphic notion of "action," represented by the Act data type.

A. Actions

The Act parameterized data type is a useful device for tying together parallel notions of "action" in roles, procs, and the runtime. If X is a Type, $Act\ X$ is the type whose constructors are

$$\mathsf{Prm} : X \to Act\ X \qquad \mathsf{Rcv} : X \to X \to Act\ X$$
$$\mathsf{Ret} : X \to Act\ X \qquad \mathsf{Snd} : X \to X \to Act\ X$$

The constructor Prm builds parameters, Ret builds return values, and Rcv and Snd builds values received and sent. These "values" will be symbolic terms, locations, or runtime values, as appropriate to the context.

B. Sorts

Symbolic terms, proc expressions, and runtime values obey a common sort discipline. The *sorts* are **chan**, **name**, **data** , **skey**, **akey**, **ikey**, and **mesg** .

C. Symbolic Terms

We begin with a set of atoms, natural numbers tagged with a constructor indicating its sort in an obvious way. We close under the operations of pairing, encryption, hashing, and quotation.

$$\begin{aligned} Term ::= {} & Atom \mid \mathsf{Sk}\ Skey \mid \mathsf{Ak}\ Akey \mid \mathsf{Ik}\ Akey \mid \\ & \mathsf{Pr}\ Term\ Term \mid \mathsf{En}\ Term\ Term \mid \\ & \mathsf{Hs}\ Term \mid \mathsf{Qt}\ \ \text{string} \\ Akey ::= {} & Sv\ n \mid Av\ n \\ Atom ::= {} & \mathsf{Ch}\ n \mid \mathsf{Tx}\ n \mid \mathsf{Dt}\ n \mid \mathsf{Nm}\ n \mid \end{aligned}$$

The *elementary* terms are the terms whose top-level constructor is *not* one of Pr, En, Hs, or Qt. A *symbolic key pair* is an ordered pair $((\mathsf{Ik}\ (\mathsf{Av}\ n)), (\mathsf{Ak}\ (\mathsf{Av}\ n)))$ consisting of a private key and a public key—in that order—which are inverses for asymmetric encryption.

The inverse function t^{-1} on algebraic terms t is not named by a constructor, but it is definable: if (t_1, t_2) is a symbolic key pair then ${t_1}^{-1} = t_2$ and ${t_2}^{-1} = t_1$; otherwise $t^{-1} = t$.

D. Roles

These are the inputs to Molly. A role is a sequence of (Act Term). Thus a role of a protocol specifies the parameters, the messages to be sent and received, and the outputs.

Our roles are essentially the roles of CPSA [24], omitting the specification there of the "uniquely originating" terms. This notion is crucial to analysis but not relevant to compiling.

E. Procs

These are the outputs of Molly, programs in an intermediate language for representing straight-line cryptographic programs. A proc can be readily translated—with the help of a suitable cryptographic library—into a conventional imperative program. A *proc* is a sequence of *statements.*

Statements: A statement is an *Event*, a *Bind*, or a *Check*.

- An *Event* is an (Act Loc). For example the event (Rcv v_1 v_2) expresses the action that a value is read from the channel stored in v_1 and stored in v_2.
- A *Bind* is of the form

$$\mathsf{Bind}\ (t, v)\ e$$

  expressing the fact that storing the value named by expression e into the location v; the symbolic term t serves as a type for the location v.
- The *Expressions* e in a Bind statement are built from locations by a set of operators mirroring the runtime operators: Pair (pairing), Frst, Scnd (projections), Encr (encryption), Decr (decryption), Hash (hashing), Quot (string constant), PubOf (public part of a private key),

Genr (a generated value) , Param (an input parameter), anf Read (a value read).

- A *Check* is an assertion: if it succeeds, computation simply continues, and if it fails, computation halts. The check statements are
  - (CSrt v s) : v_1 has the sort s
  - (CSame v_1 v_2) : the given locations have the same value
  - (CHash v_1 v_2) : v_2 is the hash of v_1
  - (CKypr v_1 v_2) : the two locs make a private/public runtime key pair[3]
  - (CQot v s) : v stores the string s

Sometimes—for instance when defining proc transcripts—we will want to focus on the Events of a proc, filtering out the the bindings and checks.

Definition 4 (Trace of a Proc): Let pr be a proc. The subsequence of Event statements of pr is called the *trace* of the proc. ///

VII. Compiler algorithm

Now we can give more detail about the compilation process introduced in Section V.

Given input role rl, we loop through the actions of the role:

- 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
- 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

A. Invariants

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

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

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

$$(\beta\ t\ v) \text{ if for some } e, (\mathsf{Bind}\ (t, l)\ e) \text{ is in pr.} \qquad (1)$$

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

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

$$\mathrm{map_R}\ \beta^{\mathsf{Act}}\ \mathit{done}\ \mathsf{pr}$$

[3]When terms are interpreted by runtime values, the public part of a key pair can be feasibly computed from the private part, though not vice-versa, and so it is feasible to check whether a pair of values makes a key pair.

Here β^{Act} is lifting of β to the Act data type and $\mathrm{map_R}$ is the natural generalization of the map function on lists when the function argument is generalized to be a relation, here β^{Act}.

These invariants lead immediately to

Lemma 1: Let pr be the result of a successful compilation.

- The role rl and the trace of pr are related as

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

- pr is saturated

These are the two properties that yield a proof of our core correctness result, the Reflecting Transcripts theorem. We next give a more formal, though still necessarily incomplete, treatment of saturation.

B. Saturation

A proc pr is saturated if it is **closed** and **justified.** We explain these in turn. The following convention will make the definitions easier to read.

Notation 1: In the inference rules below we write

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

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 certain constraints, such as equality between values, or well-sortedness of a value.

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. The rules fall into three categories:

- Elimination Rules, that decompose bindings for complex terms into bindings for their constituents (necessary for analyzing receptions)
- Introduction Rules, that generate bindings for complex terms out of bindings for their constituents (necessary to generate transmission or to build needed decryption keys, as in Example 1) and
- Check statements, functioning as runtime assertions.

Space considerations preclude treating every case of the definition; a complete treatment is in the full paper. Here are three sample closure rules, one from each category.

Pair Elimination Left and Right Rules

We apply these rules when e not a pair expression for (Pr t_1 t_2)

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

Pair Introduction Rule

We require that (Pr t_1 t_2) is in unv, and that pr has no bindings for (Pr t_1 t_2) .

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

Check Same Rule [4]

We apply this rule when t is an elementary term.

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

2) Being Justified: Intuitively, a proc pr is justified if, (i) received encryptions always have decryption keys available, and (ii) whenever $(\mathsf{Hs}\ t)$ is bound in pr then t is also bound. Formally:

Definition 5 (Justified Proc): A proc pr is *justified* if it satisfies

$$\begin{aligned}\mathsf{Bind}\ ((\mathsf{En}\ p\ k), v)\ e \wedge\ \text{non-Encryption}\ e \rightarrow \\ \exists v_1, \exists e_1, \mathsf{Bind}\ (k^{-1}, v_1)\ e_1 \\ \mathsf{Bind}\ ((\mathsf{Hs}\ t_1), v)\ e \wedge\ \text{non-Hash}\ \ e \rightarrow \\ \exists v_1, \exists e_1, \mathsf{Bind}\ (t_1, v_1)\ e_1\end{aligned}$$

///

To see the motivation for the hash justification clause, observe that there is no analog of projection or decryption for hashes. So the only way to check that a reception of shape $(\mathsf{Hs}\ t)$ is valid is to have a binding for t available so that we can hash that value and compare it to the received value.

We can now give the crucial definition of saturation.

Definition 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. ///

The process of saturation always terminates.

Theorem 7: 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 $2|(\mathsf{Pr}\ t_1\ t_2)|$.

[4]We have simplified this rule for presentation here, The actual rule deployed in Molly has an optimization that avoids a surfeit of CSame statements in the generated code

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.

3) Closure vs Being Justified: Compilation succeeds precisely when saturation succeeds at each role event, which is to say, the saturation process delivers a proc that is closed and justified. Now, closure always succeeds: the process runs until no more rules can be applied, and our termination analysis says this will eventually halt (Thm. 7) .

On the other hand, 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 keys or bodies of hashes.

So the only way compilation can fail is that we halt with a closed proc that isn't justified. We expand on this observation in Section X.

VIII. 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—presented as an equational theory—about the runtime as an axiomatic theory $\mathcal{R}$.

A. The Runtime Operators

We postulate a runtime operators corresponding to each operator in target language of procs (see Section VI-E) . For constructing values we have

`pair` `encr` `hash` `quot` `pubof` `gen`

where `pubof` computes the public part given the private part of a key pair (defined below), and `gen` is a mechanism for generating values of a given sort. We view `pubof` as a partial function over runtime values, defined only for private keys.

There are operators for destructing pairs and encryptions and an operator to return the sort of a value:

`frst` `scnd` `decr` `rtsort`

We view `frst`, `scnd`, and `decr` as partial functions, modeling the fact that they can fail when not supplied with suitable inputs.

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

B. The Axioms

a) Relating Pairing and Projection: The operations `frst` and `scnd` are the usual projections characterizing pairs.

$$\texttt{pair}\ r_1\ r_2 = r \ \leftrightarrow\ \texttt{frst}\, r \downarrow r_1 \wedge \texttt{scnd}\, r \downarrow r_2 \tag{2}$$

b) Axioms about `pubof`*:* The `pubof` partial function makes a bijection from sort **ikey** to the sort **akey** and is undefined elsewhere.

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

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

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

We will say that the ordered pair (r_1, r_2) makes a *key pair* if $\texttt{pubof}(r_1) = r_2$.

c) Relating Encryption and Decryption:

$$\texttt{encr}\ r_p\ r_{ke}\ r_e\ \leftrightarrow \texttt{decr}\ r_e\ \texttt{rtinv}\,(r_{ke}) = r_p \tag{6}$$

For convenience we have phrased this axiom in terms of the definable function `rtinv`, which is not a runtime primitive for us, but defined as follows: $\texttt{rtinv}(r_1) = (r_2)$ if

- (r_1, r_2) makes a key pair or
- (r_2, r_1) makes a key pair or
- $r_1 = r_2$ and is not of sort **ikey** or **akey**

d) Axiom about `gen`*:* The `gen` operation delivers values of the appropriate sorts.

$$\texttt{rtsort}(\texttt{gen}\ n\ srt) = srt \tag{7}$$

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

C. How the Axioms are Used

We present the above as a natural basic theory of the runtime. But in fact our main correctness theorem for the compiler, Theorem 8, requires a remarkably small number of assumptions about the runtime: indeed we do not require all of the above axioms in proving Theorem 8.

- We only use the $\rightarrow$ direction of axiom 2 concerning pairing.
- We don't use the axioms about pubof (axioms 3, 4, and 5.)
- Notably, we assume no properties, such as randomness, freshness, uniqueness, etc, about `gen` other that the fact that it generates values of the proper sort (axiom 7).

These claims are readily checked when proofs are mechanically verified: we simply edit the Coq code to reflect these weakenings and confirm that the proof of the theorem is undisturbed.

IX. Proving Transcripts are Reflected

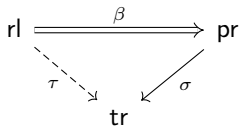
Here we describe the proof of our main correctness theorem, capturing the fact that if role rl is compiled to proc pr, then the executions of pr are reflected by the executions of rl.

Theorem 8 (Reflecting Transcripts): Let rl be a role, and suppose that rl successfully compiles to proc pr. Then any pr-compatible store-based transcript is an rl-compatible valuation-based transcript.

Proof:[Sketch] Recall the definition relating terms to locations based on the Bind statements in proc:

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

Now suppose tr is a pr-compatible store-based transcript. 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.



Formally:

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

This yields (modulo lifting these functions and relations to the Act data type) a raw transcript. It is easy to see that this transcript is in fact our original tr, and is induced by τ, and so tr is an rl-*compatible transcript*, induced by τ. To establish that tr is *valuation-based* it suffices to show that τ is a valuation. This is where the Saturation conditions on pr come into play.

We consider some representative conditions of being a valuation.

- To see that τ is functional on elementary terms: suppose we are 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

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

  By the Check Same Condition (see Section VII-B) on pr we have $(\mathsf{CSame}\ v_1\ v_2)$ in pr. Since σ respects the sameness checks of pr we have $\sigma(v_1) = \sigma(v_2)$ as desired.
- To see that 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 $\texttt{pair}\, r_1 r_2 = r$. 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. We consider the (slightly more complicated) latter case. We use the Pair Elimination

inference rules (Section VII-B). Closure under these rules yields locations v_1 and v_2 such that

$$\mathsf{Bind}\ (t_1, v_1)\ (\mathsf{Frst}\ v)\ \text{and}\ \mathsf{Bind}\ (t_2, v_2)\ (\mathsf{Scnd}\ v)$$

Set $r_1 = \sigma v_1$ and $r_2 = \sigma v_2$. Since σ is a pr-store, it respects Frst and Scnd,

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

that is,

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

Now we simply apply one of our axioms about the runtime, namely

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

In the remaining case, where e is a pair expression for $(\mathsf{Pr}\ t_1\ t_2)$, we will use the Pair Introduction rule of Section VII-B; however, space precludes giving more detail.

- To see that 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, item 1 of Definition 1.
  By definition of τ we have v and e such that

$$\mathsf{Bind}\ ((\mathsf{En}\ p\ k_e), v)\ e \in \mathsf{pr} \tag{8}$$
$$\sigma(v) = r_e. \tag{9}$$

  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

$$\mathsf{Bind}\ ((\mathsf{En}\ p\ k_e), v_e)\ (\mathsf{Encr}\ v_p\ v_k) \in \mathsf{pr} \tag{10}$$

     for some v_p and v_k. We establish the encr condition, *i.e.*, that there exists r_p, r_{ke} such that
     - $(\tau\ p\ r_p)$
     - $(\tau\ k_e\ r_{ke})$
     - $\mathtt{encr}\ r_p\ r_k\ r_e$

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

$$\mathsf{Bind}\ (t_p, v_p)\ e_p \in \mathsf{pr} \tag{11}$$
$$\mathsf{Bind}\ (t_k, v_k)\ e_k \in \mathsf{pr} \tag{12}$$

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

$$(\tau\ p\ r_p) \tag{13}$$
$$(\tau\ k_e\ r_{ke}). \tag{14}$$

     Since σ is a pr-store, it respects respects Encr and so we have, by (10),

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

     holds at runtime which is to say

$$\mathtt{encr}\, r_p\ r_k\ r_e. \tag{15}$$

     The encr condition follows from (13), (14), and (15)
  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 (8) there are v_{kd} and e_{kd} such that

$$\mathsf{Bind}\ ((k_e)^{-1}, v_{kd})\ e_{kd} \in \mathsf{pr} \tag{16}$$

     Set r_{kd} to be (σv_{kd}), thus

$$\tau(k_e)^{-1} r_{kd} \tag{17}$$

     By the Decryption Condition applied to (8) and (9) we have

$$\mathsf{Bind}\ (p, v_p)\ (\mathsf{Decr}\ v\ v_{kd}) \in \mathsf{pr} \tag{18}$$

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

$$\tau\ p\ r_p \tag{19}$$

     Since σ respects Decr, we have, by (18),

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

     which is to say

$$\mathtt{decr}\ r_e\ r_{kd} \downarrow r_p \tag{20}$$

     The decryption condition follows from (17), (19), and (20)

///

Remark

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

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

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

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

  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

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

instead of going to the trouble of defining the disjunctive condition in item 1 of Definition 1? 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 (necessary!) 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.

X. When Does Compilation Fail?

In this section we explore 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 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.

To state this precisely, first let us say that 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.

Next let us add two natural tweaks to the traditional Dolev-Yao system: (i) we may derive, without hypotheses, (Qt s) for any string s, and (ii) we may derive the public part of a key pair from the corresponding private part.

Then we have the following relationship between derivation and compilation.

Theorem 9: Suppose pr is a proc generated from a role rl.

- 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.
- When pr is closed, then 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: The proof follows naturally from the observation that each of the closure rules corresponds to a Dolev-Yao inference rule when the information about locations and proc-expressions is erased.

A. Executability

Several authors (*e.g.*, [12], [8], [13]) have defined notions of *executability* of a protocol, statically-checkable properties that give confidence that a protocol can be run to completion. In our situation we can connect these ideas to the success or failure of role compilation (a static property of roles).

As observed in Section VII-B, the only way compilation can fail is that we halt with a closed proc that isn't justified. This means (cf. Definition 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.

But in the first case, the term (En p k) is derivable from the terms obtained in rl but the term k^{-1} is not derivable from the terms obtained in rl. Similarly, in the second case the term (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 in Molly is reflected by the existence of terms from the role that are required in order for the proc to be able to construct statements it needs but cannot be Dolev-Yao derived.

XI. Related Work

Molly's goals are complementary to those of computer-aided cryptography, a growing area of research that applies formal verification to the design, analysis, and implementation of cryptography. Barbosa et al [7] includes a systematic overview of that literature as of 2021; the Last Yard framework is a recent "unified foundational Coq framework for the end-to-end verification of high-speed cryptography" [20].

As discussed in the introduction, our goals are amplified by those of computer-aided cryptography. We assume the existence of libraries for cryptographic primitives, and we generate code for the processes of message transmission and reception that will be linked with such libraries. Our focus is on bridging the gap between protocol narrations in a semi-formal style and protocol descriptions that are more formal.

Correspondingly, our notion of correctness is independent of the correctness of the design or the implementation of cryptographic algorithms (and certainly independent of higher-level questions such as whether a protocol validates certain security goals). Rather we focus on the specific claim that the code Molly generates implements the processes of transmitting and receiving bitstrings at correctly, according to the specification embodied in the given symbolic role. This is the content of Theorem 8.

The fact that the runtime assumptions we require for this theorem are so modest VIII-B suggest that our results should integrate seamlessly with computer-aided cryptography frameworks.

The development of verified compilers is by now a well-established area, for conventional languages (CompCert [23] is an exemplar here) as well as domain-specific languages (*e.g.*, [33]).

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 [25], to generic intermediate languages [16] [27], [3], to Multiset Rewriting [21] [22], to symbolic representations of principals' knowledge [26] [8] (annotated with unifiability conditions in [13]), and to Pi-calculus and variants [11] [12] [10].

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 [15] 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 runtime values.

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 [12] 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 VII-B 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 functioning as runtime values.

Arquint et al [4], [5] 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 [31]. Then (quoting [4]) "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 8, with logical properties standing in for runtime values.

We now turn to the category of projects that are primarily focused on generating implementations from protocol specifications. Although the work in [3] is not principally focused in this way, that paper reports a translation from its intermediate language SPS into JavaScript.

Tobler and Hutchinson [37] 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 [6] developed Expi2Java, which translates models written in the Spi calculus [2] 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 [28], [29] 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 [9] verification that security goals are met by the Java code.

The JavaSPI tool of Sisto, Copet, Avalle and Bronte [36] starts with code in a fragment of the language that corresponds to applied pi-calculus [1]. 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 [35] 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 (similar to the approach in Pnueli et al [32]). 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 runtime values, 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.

XII. Conclusion and 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 digital signatures and richer notions of tupling. A more significant extension of the current work will connect it with protocol *analysis*. Specifically we will integrate Molly into the CPSA ecosystem, so that a protocol designer, having established some symbolic-level security goals for her protocol using a CPSA analysis, can generate implementations of the protocol roles satisfying those goals.

One would also like to know that a symbolic security goal established by CPSA holds true of the joint *runtime* actions of a set of participants if they all use Molly-generated code linked against strong cryptographic libraries, including CCA2 encryption and existentially unforgeable signatures. A few additional properties are needed, such as disjointness assumptions for different operators and elementary sorts. With those in place, we conjecture that any observed execution is, with overwhelming probability, an instance of a symbolic execution that CPSA has considered.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE symposium on security and privacy (SP)*, pages 777–795. IEEE, 2021.

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] Daniel J. Dougherty and Joshua D. Guttman. Molly: A Verified Compiler for Cryptoprotocol Roles. *arXiv e-prints*, page arXiv:2311.13692, November 2023.

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

[20] Philipp G Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. The last yard: Foundational end-to-end verification of high-speed cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 30–44, 2024.

[21] 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.

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

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

[24] Moses D. Liskov, John D. Ramsdell, Joshua D. Guttman, and Paul D. Rowe. *The Cryptographic Protocol Shapes Analyzer: A Manual for CPSA 4*. The MITRE Corporation, 2023. https://github.com/mitre/cpsa/blob/master/doc/cpsa4manual.pdf.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4*, pages 151–166. Springer, 1998.

[33] 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.

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

[35] 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.

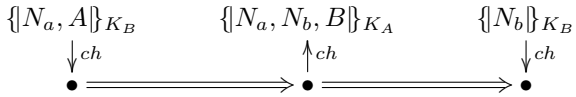
[36] Riccardo Sisto, Piergiuseppe Bettassa Copet, Matteo Avalle, and Alfredo Pironti. Formally sound implementations of security protocols with JavaSPI. *Formal Aspects of Computing*, 30:279–317, 2018.

[37] 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.

APPENDIX

NSL IN MOLLY

We now have enough information about the compiler to present the example from Section II fully realized, discussing some of the techniques of the compiler in detail. Here is the responder role of the Needham-Schroeder-Lowe protocol was presented in strand notation:



To render this role in the input language of Molly we first express the data in an ASCII syntax.

- we write the channel ch as `(Ch 0)`
- we write participant A's name as `(Nm 0)`
- we write participant B's name as `(Nm 1)`
- we write A's public key K_A as `(Ak (Av 0))`
- we write B's public key K_B as `(Ak (Av 1))`
- we write B's private key ${K_B}^{-1}$ as `(Ik (Av 1))`
- we write A's nonce N_A as `(Tx 0)`
- we write B's nonce N_B as `(Tx 1)`

We write the actions of the role, the two receptions and the transmission, as `Rcv` and `Snd` events, respectively. The parameters are listed explicitly in the five `Prm` expressions, and the two receptions with the intervening transmissions are given next. At the end, the Molly role presentation is explicit about what the role returns (in contrast to the strand notation) by listing two return events.

Thus the NSL responder strand is captured as the following Molly role.

NSL responder as Molly input

```
Prm (Ch 0)
Prm (Nm 0)
Prm (Nm 1)
Prm (Ak (Av 0))
Prm (Ak (Av 1))
Prm (Ik (Av 1))

Rcv (Ch 0)
  (En (Pr (Tx 0) (Nm 0))
     (Ak (Av 1)))

Snd (Ch 0)
  (En (Pr (Tx 0) (Pr (Tx 1) (Nm 1)))
     (Ak (Av 0)))

Rcv (Ch 0)
  (En (Tx 1)
     (Ak (Av 1)))

Ret (Tx 0)
Ret (Tx 1)
```

When Molly compiles this role we have the following proc as output. The reader will probably want to start with the guide that follows.

NSL responder as Molly output

```
1
2  Bind (Tx 1, L 1) (Genr 1 Text);
3  Csrt (L 1) Text;
4  Evnt (Prm (L 2));
5  Bind (Ch 0, L 2) (Param 1);
6  Csrt (L 2) Chan;
7  Evnt (Prm (L 3));
8  Bind (Nm 0, L 3) (Param 2);
9  Csrt (L 3) Name;
10 Evnt (Prm (L 4));
11 Bind (Nm 1, L 4) (Param 3);
12 Csrt (L 4) Name;
13 Bind (Pr (Tx 1) (Nm 1), L 5)
14   (Pair (L 1) (L 4));
15 Evnt (Prm (L 6));
16 Bind (Ak (Av 0), L 6) (Param 4);
17 Csrt (L 6) Akey;
18 Evnt (Prm (L 7));
19 Bind (Ak (Av 1), L 7) (Param 5);
20 Csrt (L 7) Akey;
21 Bind (En (Tx 1) (Ak (Av 1)), L 8)
22      (Encr (L 1) (L 7));
23 Evnt (Prm (L 9));
24 Bind (Ik (Av 1), L 9) (Param 6);
25 Csrt (L 9) Ikey;
26 Ckypr (L 9) (L 7);
27 Evnt (Rcv (L 2) (L 10));
28 Bind
29   (En (Pr (Tx 0) (Nm 0))
30       (Ak (Av 1)), L 10)
31   (Read 1);
32 Bind (Pr (Tx 0) (Nm 0), L 11)
33      (Decr (L 10) (L 9));
34 Bind (Tx 0, L 12) (Frst (L 11));
35 Bind (Nm 0, L 13) (Scnd (L 11));
36 Bind (Pr (Tx 0) (Pr (Tx 1) (Nm 1)),
37                    L 14)
38      (Pair (L 12) (L 5));
39 Bind
40   (En (Pr (Tx 0) (Pr (Tx 1) (Nm 1)))
41      (Ak (Av 0)), L 15)
42    (Encr (L 14) (L 6));
43 Csrt (L 12) Text;
44 Csame (L 3) (L 13);
45 Evnt (Snd (L 2) (L 15));
46 Evnt (Rcv (L 2) (L 16));
47 Bind (En (Tx 1) (Ak (Av 1)), L 16)
48       (Read 2);
49 Bind (Tx 1, L 17) (Decr (L 16) (L 9));
50 Csame (L 1) (L 17);
51 Evnt (Ret (L 12));
52 Evnt (Ret (L 1))
```

Notes on the proc

- Lines 2 and 3 generate B's nonce (Tx 1) and do a checksort. For simplicity of design, Molly emits code for the generation of such local values in an initialization phase, even though (as in this case) these values may not be required till later in the execution..
- Line 4 starts processing of 1st parameter: line 5 stores the parameters value in a fresh location, (L 2), and line 6 is code for the runtime check that this parameter really is a channel.
- The five other parameters are processed similarly, starting at lines 7, 10, 15, 18, and 23 respectively.
- Lines 27 and 46 respectively initiate the two receptions; line 45 is the transmission event.
- Most of the complexity of compilation lies in the processing of receptions; let us unpack the proc code for the reception of $\{|A, N_a|\}_{K_B}$ at lines 27 through 35.
  - Line 27 itself records the fact that a reception happens on the channel at location (L 2) and the value received is stored at the fresh location (L 10).
  - The Bind at line 28 records the fact that the symbolic term corresponding to the value in (L 10) is $\{|A, N_a|\}_{K_B}$, which is (En (Pr (Tx 0) (Nm 0)) (Ak (Av 1)), L 10) (Read 1) in Molly syntax, and that this value is expressed by the first Read expression.
  - Line 32 is the start of the process of *validating* this reception.
    - we (attempt to) bind the body (Pr (Tx 0) (Nm 0)) of the encryption to a location (L 11) by decrypting (L 10) by the key stored in (L 9). We expect that (L 9) holds the decryption key for this encryption because in previous lines (18 and 23 we bound (L 7) to the public key of B, and (L 9) to its inverse. Note that this Decr operation can fail at runtime, for example if the runtime value in (L 10) is not an encryption, or if the value in (L 9) is not a suitable decryption key. We also have a runtime check, in line 26, that the values (L 7) and (L 9) do indeed form a key pair.
    - In a similar way, lines 34 and 35 ensure that the body of our encryption is really a pair, we performing the two projection operations Frst and Scnd and binding the resulting values to new locations and recording the expected symbolic-term associations.
- Lines such as 13, and lines 38 through 44 have a different character than those arising out of "immediate" requirements. They are part of the saturation process, but they generate bindings for terms that *could be* required later.
  For instance, the transmission in line 45 has the value to be sent already constructed and bound to a location. Why do we do this advance construction of values, rather than building them only as we need them? This just-in-case strategy (the opposite of a just-in-time strategy) is actually required, as we observed in the *Saturation is not*

Syntax-Directed paragraph of Section V, (see Example 1). The key point is that the process of fully *destructing* input values to test that they are suitable can involving *constructing* values from data already received.
Molly schedules these constructions eagerly, so that by the time a message is to be transmitted, it is already bound to a location; this is why transmission events themselves such as line 45 are so simple.