Cryptographic Protocol Analysis and Compilation Using CPSA and Roletran

John D. Ramsdell(✉)

The MITRE Corporation, Bedford, MA 01730, USA
ramsdell@mitre.org

Abstract. The Cryptographic Protocol Shapes Analyzer CPSA determines if a cryptographic protocol achieves authentication and secrecy goals. It can be difficult to ensure that an implementation of a protocol matches up with what CPSA analyzed, and therefore be sure the implementation achieves the security goals determined by CPSA.

Roletran is a program distributed with CPSA that translates a role in a protocol into a language independent description of a procedure that is easily translated into an existing computer language. This paper shows how we ensure the procedure produced by Roletran is faithful to strand space semantics and therefore achieves the security goals determined by CPSA.

Real implementations of cryptographic functions make use of probabilistic encryption, but CPSA will conclude that two encryptions are the same if they are constructed with the same plaintext and key. The paper concludes by showing how we ensure that executions of generated code that make use of probabilistic encryption achieve the goals determined by CPSA.

1 Introduction

The Cryptographic Protocol Shapes Analyzer (CPSA) [8] attempts to enumerate all essentially different executions possible for a cryptographic protocol. We call them the shapes of the protocol. Naturally occurring protocols have only finitely many, indeed very few shapes. Authentication and secrecy properties are easy to determine from them, as are attacks and anomalies.

For each input problem, the CPSA program is given some initial behavior, and it discovers what shapes are compatible with it. Normally, the initial behavior is from the point of view of one participant. The analysis reveals what the other participants must have done, given the participant's view. The search is complete, i.e. we proved every shape can in fact be found in a finite number of steps, relative to a procedural semantics of protocol roles [7].

This paper is dedicated to Joshua Guttman in gratitude for all the wonderful collaborations we shared throughout our careers. From the first rigorous verification of the implementation of a programming language in actual use (Scheme via the VLISP *project* [6]*), to cryptographic protocol analysis (*CPSA*), it has been a joy to work with you.*

D. Dougherty et al. (Eds.): Guttman Festschrift, LNCS 13066, pp. 355–369, 2021.
https://doi.org/10.1007/978-3-030-91631-2_20

When we say a role has procedural semantics, we mean that there exists a program that implements the intent of the specified role. Until now, establishing the correspondence between a CPSA role and its implementation has been informal. It requires a programmer that is well versed in the semantics of CPSA. As the messages used in roles become more complex, the likelihood of errors in the correspondence increases, even when employing the best programmer/CPSA expert. The Roletran compiler automates the translation of a CPSA role into a procedure that is easily translated into the source for an existing programming language, in our case, Rust. It uses the same algorithms implemented in CPSA to ensure a faithful translation. But how do we know its translations are correct?

Section 4 presents the abstract semantics of procedures used to guide our implementation of a runtime system for Roletran generated programs. It includes a definition of correctness, Definition 8, that precisely defines whether the output of Roletran correctly implements the role it is given.

The semantics presented in Sect. 4 has been specified in Coq [1]. An attempt was made to specify the Roletran compiler as a function in Coq and prove that every output of Roletran correctly implements the role it is given. However, the proofs turned out to be too complex and challenging, and the attempt was abandoned.

As a fallback, one can present Coq with the runtime semantics, a role, and procedure, and when the procedure is the output of Roletran, Coq will automatically prove it correctly implements the role. The Coq automation succeeds on protocols of substantial size. Thus for high-assurance applications, we provide a means to validate compiler input/output pairs in lieu of verifying the compiler algorithm.

There is one loose end in what might seem to be a tidy story at this point. Real implementations of cryptographic functions make use of probabilistic encryption. This means that there may be several bit patterns that correspond to one encryption term in CPSA. If the compiler generates code that asserts that two encryptions are equal, the assertion might fail at runtime if the two encryptions differ only because of the randomness used to generate them. To explore these issues, a more concrete semantics has been defined that models randomness in encryptions. The paper concludes by showing that

1. the concrete semantics is faithful to the abstract semantics, in that for every run of the concrete semantics, there is a corresponding run of the abstract semantics, and
2. the concrete semantics is adequate with respect to the abstract semantics, in that for every run of the abstract semantics and choice of random values, there is a corresponding run of the concrete semantics.

Therefore, probabilistic encryption is handled correctly.

* * *

The Roletran compiler and supporting Coq proofs were written by the author and the sources are available on GitHub [8]. The sources contain about 5800 lines of Coq scripts and the Intro module presents an overview of the work.

There have been a variety of systems that compile high-level descriptions of protocols into executable code [2,5]. To our knowledge, this is the first example of a compiler that uses the input of a cryptographic protocol analyzer as its sole input and honors its semantics.

Notation. A finite sequence f is a function from an initial segment of the natural numbers. The length of f is $|f|$, and $f = \langle f(0), \ldots, f(n-1) \rangle$ for $n = |f|$. The sequence $x :: f$ is $\langle x, f(0), \ldots, f(n-1) \rangle$. The concatenation of sequences f_1 and f_2 is $f_1 \frown f_2$.

If S is a set, then S^* is the set of finite sequences over S, and S^+ is the non-empty finite sequences over S. If S is a finite set, then $\overrightarrow{S}$ is some injective sequence that is onto S. That is, it is a sequence that contains every element in S without duplicates.

Suppose $g : X \rightharpoonup Y$ is a finite partial function.

$$g[x \mapsto y](z) = \begin{cases} y & \text{if } z = x, \\ g(z) & \text{otherwise.} \end{cases}$$

We use $\emptyset$ to denote the finite partial function that has an empty domain.

2 Message Algebras

This section describes the formalism on which CPSA message algebras are based. The parameters to an algebra are:

1. a set of messages Alg. The set of messages Alg is the carrier set (or domain) of a term algebra.
2. a set of basic values $\mathsf{BV} \subset \mathsf{Alg}$. Keys and nonces are examples of basic values.
3. a *carried by* relation $\sqsubseteq \subseteq \mathsf{Alg} \times \mathsf{Alg}$. Intuitively, a message t_0 is carried by t_1 if it is possible to extract t_0 from t_1 by someone who knows the relevant decryption keys.

Example Message Algebra. The signature of one possible order-sorted [4] message algebra is in Fig. 1. The algebra is the simplification of the CPSA message algebra used by the examples in this paper.

In an order-sorted algebra, each variable x has a unique sort S. The *declaration* of x is $x : S$.

The algebra of interest is the order-sorted quotient term algebra generated by a set of declarations X. The message algebra Alg_X is the carrier set for sort M. The set of basic values BV_X is the union of the carrier sets for sorts A, S, and D. The carrier set for sort A contains the algebra's asymmetric key pairs. We write $t : S$ to say that term t is in the carrier set of sort S.

A variable has no intrinsic sort associated with it. The declarations that generate an algebra determine the sort of variables that occur within terms of the algebra. A variable declared to be of sort M is called a *message variable*.

Sorts:	M, A, S, D		
Subsorts:	$\mathsf{A} < \mathsf{M}$, $\mathsf{S} < \mathsf{M}$, $\mathsf{D} < \mathsf{M}$		
Operations:	$(\cdot,\cdot)$	$: \mathsf{M} \times \mathsf{M} \to \mathsf{M}$	Pairing
	$\{\!\|\cdot\|\!\}_{(\cdot)}$	$: \mathsf{M} \times \mathsf{M} \to \mathsf{M}$	Encryption
	$\#$	$: \mathsf{M} \to \mathsf{M}$	Hash
	$(\cdot)^{-1}$	$: \mathsf{M} \to \mathsf{M}$	Key inverse
	$\tau_0, \tau_1, \ldots$	$: \mathsf{M}$	Tag constants
Equations:	$(x^{-1})^{-1} = x$ for $x : \mathsf{A}$; $x^{-1} = x$ otherwise		

Fig. 1. Simple crypto algebra signature

The Simple Crypto Algebra is interesting because like CPSA's message algebra, any message can be used as a key when constructing an encryption, with the exception of a message variable. The reason for the exception is that message variable x could be unified with any basic value, and so what equation applies to x^{-1}?

Each element of the message algebra is a set of terms. The canonical representative of each element is the term with the fewest number of occurrences of the inverse operation $(\cdot)^{-1}$. Thus when x is a variable, the canonical representative of the algebra element that contains

$$((x^{-1})^{-1})^{-1}$$

is x^{-1} if $x : \mathsf{A}$, and x otherwise. Message t_0 *occurs* in t_1 iff the canonical representative of t_0 is a subterm of the canonical representative of t_1. In what follows, we conflate each algebra element with its canonical representative.

Definition 1 (Encryption free terms). *Term t is* encryption free*, written enc_free t, iff no encryption term occurs in t.*

A message t_0 is *carried by* t_1, written $t_0 \sqsubseteq t_1$, if t_0 can be derived from t_1 given the right set of keys. That is: $\sqsubseteq$ is the smallest reflexive, transitive relation such that

$$t_0 \sqsubseteq (t_0, t_1), \quad t_1 \sqsubseteq (t_0, t_1), \quad \text{and} \quad t_0 \sqsubseteq \{\!|t_0|\!\}_{t_1}.$$

3 Strand Spaces with Channels

The foundation of this work is a version of strand spaces in which messages are transmitted over channels. This change facilitates the translation of a role into code by adding a natural handle for performing input and output in generated code.

Recall that a strand space [9] is a finite map from a local session of a protocol, called a strand, to its behavior, called a trace. The addition of channels changes the standard definition of a trace, but otherwise leaves the basic definitions of strand space theory unchanged.

A channel is a variable of sort C. For $h : \mathsf{C}$ and $t : \mathsf{M}$, $[h, t]$ associates message t with channel h, and is called a *channeled message*. The additions to a message signature required to support channels follow.

Extra Sorts: C, CM
Operation: $[\cdot, \cdot] : \mathsf{C} \times \mathsf{M} \to \mathsf{CM}$ Channeled messages

The sort associated with a channeled message is CM. The carrier set for that sort is $\overline{\mathsf{Alg}}_X$. Variables of sort CM are not allowed in X. The carrier set for sort C is Chn_X. Let $\widehat{\mathsf{Alg}_X} = \mathsf{Alg}_X \cup \mathsf{Chn}_X$.

Traces and Roles. The behavior of a strand, its *trace*, is a finite non-empty sequence of *events*. An *event* is either a *channeled message transmission* or a *channeled message reception*. An event transmitting $m \in \overline{\mathsf{Alg}}_X$ is written as $+m$; and an event receiving channeled message m is written as $-m$. If $e = \pm[h, t]$ is an event, then $msg(e) = t$. The set of traces over $\overline{\mathsf{Alg}}_X$ is $(\pm\overline{\mathsf{Alg}}_X)^+$.

A message t *originates* in trace c at index i iff $c(i) = +[h, t_1]$, $t \sqsubseteq t_1$, and for all $j < i$, $t \not\sqsubseteq msg(c(j))$. A message t *uniquely originates* in strand space Θ iff it originates in exactly one trace in Θ. A message t is *non-originating* in strand space Θ iff it originates in no trace in Θ.

Structure $\mathsf{r}_X(c, i, o, u)$ is a *role* when

1. c is a trace in $(\pm\overline{\mathsf{Alg}}_X)^+$,
2. each variable declared in X occurs in c,
3. $i \in (\mathsf{BV}_X \cup \mathsf{Chn}_X)^*$ is a sequence of basic values and channels that specify the inputs to the role,
4. $o \in \mathsf{Alg}_X^*$ is a sequence of terms that specify the outputs of the role, and
5. $u \subseteq \mathsf{BV}_X$ is a set of basic values that originate in c.

The elements of i and o are a sequence because the order matters when generating a procedure from the role. Elements of u are freshly generated when the compiled role executes.

Executions. An execution $\mathsf{e}_Y(c, i, o, u)$ is similar to a role except that its uniquely originating values are a sequence, not a set. The semantics of an execution requires that the fresh values be presented in the order in which they are consumed. Otherwise, the components of an execution must satisfy the same constraints. Let $\phi : \widehat{\mathsf{Alg}_X} \to \widehat{\mathsf{Alg}_Y}$ be a homomorphism, and $\bar{\phi}$ be the extension of ϕ to traces and sequences of terms in the obvious way.

Definition 2 (Run of a role). *Execution* $\mathsf{e}_Y(c', i', o', u')$ *is a* run *of role* $\mathsf{r}_X(c, i, o, u)$ *iff there exists a homomorphism* ϕ *such that* $\bar{\phi}(c) = c'$, $\bar{\phi}(i) = i'$, $\bar{\phi}(o) = o'$, $\bar{\phi}(\boldsymbol{u}) = u'$, *and* $\boldsymbol{u}$ *is some sequence that contains the elements in* u.

The strand spaces model of a protocol execution is a bundle. A bundle adds a communication relation to a strand space, and constraints that ensure that causality is respected in that every received message is transmitted previously

in the bundle. In strand spaces, a Dolev-Yao adversary [3] is modeled by strands in a bundle along with the strands derived from the protocol being analyzed.

CPSA does not represent executions using bundles or adversarial behavior using strands. Instead, it uses a skeleton to represent a collection of bundles. A skeleton has a strand space, an ordering relation between events, and some origination assumptions that must be satisfied by the strand space of the skeleton. A bundle is modeled by a skeleton if it contains all of the structure specified by the skeleton, in other words, there is a homomorphism from the skeleton into the bundle.

For each input problem, CPSA is given some initial behavior, and it discovers what shapes are compatible with it. A shape is a special kind of skeleton in that it contains enough protocol behavior to explain all message receptions in the presence of adversarial behavior, and it is minimal in that if there is a homomorphism from another skeleton to the shape, then there is a homomorphism from the shape to the other skeleton.

To describe the executions of a protocol, each strand in a skeleton must be an instance of some role in the protocol, which is defined to mean there is a homomorphism from the role to the strand. The definition of a run of a role codifies that link for procedure execution semantics.

3.1 Unilateral Protocol Example

The Unilateral Protocol is a very simple authentication protocol. It consists of two roles, an initiator and a responder. The initiator encrypts a freshly chosen nonce using the public key of the responder and sends it. The responder decrypts the encryption it receives using its private key, and transmits the plaintext. If the initiator receives the nonce it sent unencrypted, it concludes it is communicating with a responder that possesses the corresponding private key, assuming the private key has not been compromised. In the notation presented above, the protocol is specified as follows.

Example 3 (Unilateral Protocol)

$$\mathsf{init} = \mathsf{r}_{h:\mathsf{C},n:\mathsf{D},k:\mathsf{A}}(\langle +[h, \{|n|\}_k], -[h, n]\rangle, \langle h, k\rangle, \langle n\rangle, \{n\})$$
$$\mathsf{resp} = \mathsf{r}_{h:\mathsf{C},n:\mathsf{D},k:\mathsf{A}}(\langle -[h, \{|n|\}_k], +[h, n]\rangle, \langle h, k^{-1}\rangle, \langle n\rangle, \{\})$$

Both the initiator and the responder use a message algebra generated by a channel h, a datum n, and an asymmetric key k. The trace of the initiator contains two events, a channeled message transmission followed by a channeled message reception. The inputs to the initiator are a channel and the public part of a key pair. The inputs to the responder are a channel and the private part of a key pair. The outputs produced by both roles is the single nonce n. The initiator freshly generates nonce n, and the responder freshly generates nothing.

CPSA determines that if an instance of an initiator role runs to completion, and the private part of the key pair is not compromised, i.e. is non-originating, then there must have been a corresponding run of the responder role that agrees with the initiator on the values of the nonce and the public key.

3.2 Channel Assumptions

With the addition of channels to CPSA, skeletons now include additional kinds of assumptions besides origination assumptions. A channel can be assumed to be authenticated and/or confidential. In a bundle, when a channel is authenticated, the adversary is not allowed to transmit a message on the channel, and when it is confidential, the adversary is not allowed to received a message on the channel. The addition of channel assumptions allows interesting new analyses of protocols, but does not impact Roletran, so it will not be further discussed.

4 Abstract Execution Semantics

Roletran generates a procedure for each role in a protocol. To build an executable program, the procedure is trivially translated into source code for an existing programming language, in our case Rust. The code is compiled and linked with a runtime system. The implementer of the program provides a main routine that invokes the procedure with inputs that must be compatible with inputs of the translated role. We trust the implementor to do so.

When the program executes, it goes through state changes associated with each statement generated by Roletran. The abstract execution semantics specifies an abstract view of properties of the states that must be preserved in order to be in compliance with the execution semantics stated in the previous section.

When the compiled translation of a role is executing, the runtime system for the source language maintains a binding between program variables and binary objects that represent message fragments. The abstract execution semantics models these bindings with a map from program variables to terms in the message algebra. This map is called an *environment.* The implementor of the runtime library must ensure that each binary object naturally abstracts into the corresponding term in the message algebra as specified by the current environment.

A runtime system for a program provides two more capabilities, support for sending and receiving messages on channels, and freshly generating random values. To model freshly generating random values, the abstract execution semantics maintains a sequence of basic values that is the source of randomness. Initially it is the sequence of uniquely originating values in an execution. The implementor of the runtime library must ensure each binary object it creates naturally abstracts into the corresponding term in the message algebra as specified by the abstract execution semantics.

To model messaging on channels, the abstract execution semantics maintains a trace that initially is the trace in the execution. The implementor of the runtime library must ensure each binary object transmitted or received naturally abstracts into the corresponding event over the message algebra as specified by the abstract execution semantics.

$$
\begin{array}{lll}
ae & : & V \rightharpoonup \widehat{\mathsf{Alg}_Y} \text{ environment} \\
& \times & (\pm\overline{\mathsf{Alg}_Y})^* \text{ input trace} \\
& \times & \mathsf{Alg}_Y^* \text{ input fresh values} \\
& \times & \mathcal{E} \text{ expression} \\
& \times & \mathsf{Alg}_Y \text{ value} \\
& \times & (\pm\overline{\mathsf{Alg}_Y})^* \text{ output trace} \\
& \times & \mathsf{Alg}_Y^* \text{ output fresh values}
\end{array}
$$

$$ae(E, c, u, \ulcorner \mathsf{quot}(\tau) \urcorner, \tau, c, u) \tag{1}$$

$$\frac{E(v_1) = t_1 \qquad E(v_2) = t_2}{ae(E, c, u, \ulcorner \mathsf{pair}(v_1, v_2) \urcorner, (t_1, t_2), c, u)} \tag{2}$$

$$\frac{E(v_1) = t_1 \qquad E(v_2) = t_2}{ae(E, c, u, \ulcorner \mathsf{encr}(v_1, v_2) \urcorner, \{\!|t_1|\!\}_{t_2}, c, u)} \tag{3}$$

$$\frac{E(v_1) = t_1}{ae(E, c, u, \ulcorner \mathsf{hash}(v_1) \urcorner, \#t_1, c, u)} \tag{4}$$

$$\frac{E(v_1) = (t_1, t_2)}{ae(E, c, u, \ulcorner \mathsf{frst}(v_1) \urcorner, t_1, c, u)} \tag{5}$$

$$\frac{E(v_1) = (t_1, t_2)}{ae(E, c, u, \ulcorner \mathsf{scnd}(v_1) \urcorner, t_2, c, u)} \tag{6}$$

$$\frac{E(v_1) = \{\!|t_1|\!\}_{t_2} \quad E(v_2) = {t_2}^{-1} \quad \mathit{enc\_free}\, {t_2}^{-1}}{ae(E, c, u, \ulcorner \mathsf{decr}(v_1, v_2) \urcorner, t_1, c, u)} \tag{7}$$

$$\frac{E(v_1) = h}{ae(E, -[h, t] :: c, u, \ulcorner \mathsf{recv}(v_1) \urcorner, t, c, u)} \tag{8}$$

$$ae(E, c, t :: u, \ulcorner \mathsf{frsh} \urcorner, t, c, u) \tag{9}$$

Fig. 2. Abstract execution expression semantics

The output of the compiler is an executable procedure $\mathsf{x}(p, s)$, where p is a sequence of parameters and s is a sequence of statements. Each parameter is a program variable and its type, and is associated with an input when the procedure is invoked. A type is one of $\mathbb{M}$, $\mathbb{A}$, $\mathbb{I}$, $\mathbb{S}$, $\mathbb{D}$, and $\mathbb{C}$.

The code generated by the compiler is a sequence of statements. Let $\mathcal{V}$ be the syntactic category for program variables. The syntax of a statement is

$$
\begin{array}{l}
\mathcal{S} ::= \mathcal{V} : \mathcal{T} \leftarrow \mathcal{E} \mid \mathcal{V} \approx \mathcal{V} \mid \mathsf{invp}(\mathcal{V}, \mathcal{V}) \mid \mathsf{send}(\mathcal{V}, \mathcal{V}) \mid \mathsf{return}(\mathcal{V}^*) \\
\mathcal{T} ::= \mathbb{M} \mid \mathbb{A} \mid \mathbb{I} \mid \mathbb{S} \mid \mathbb{D} \mid \mathbb{C} \\
\mathcal{E} ::= \mathsf{quot}(\tau) \mid \mathsf{pair}(\mathcal{V}, \mathcal{V}) \mid \mathsf{encr}(\mathcal{V}, \mathcal{V}) \mid \mathsf{hash}(\mathcal{V}) \\
\quad \mid \mathsf{frst}(\mathcal{V}) \mid \mathsf{scnd}(\mathcal{V}) \mid \mathsf{decr}(\mathcal{V}, \mathcal{V}) \mid \mathsf{recv}(\mathcal{V}) \mid \mathsf{frsh}
\end{array}
$$

$$
\begin{array}{lll}
as & : & V \rightharpoonup \widehat{\mathsf{Alg}_Y} \quad \text{input environment} \\
& \times & (\pm\overline{\mathsf{Alg}_Y})^* \quad \text{input trace} \\
& \times & \mathsf{Alg}_Y^* \quad \text{input fresh values} \\
& \times & \mathcal{S} \quad \text{statement} \\
& \times & V \rightharpoonup \widehat{\mathsf{Alg}_Y} \quad \text{output environment} \\
& \times & (\pm\overline{\mathsf{Alg}_Y})^* \quad \text{output trace} \\
& \times & \mathsf{Alg}_Y^* \quad \text{output fresh values}
\end{array}
$$

$$
\frac{ae(E, c_1, u_1, x, t, c_2, u_2) \quad chk(t, k)}{as(E, c_1, u_1, \ulcorner v : k \leftarrow x \urcorner, E[v \mapsto t], c_2, u_2)} \tag{10}
$$

$$
\begin{array}{ll}
chk(t, \mathbb{M}) & \text{always true} \\
chk(t, \mathbb{A}) & \text{iff } t \text{ is a variable of sort } \mathsf{A} \\
chk(t, \mathbb{I}) & \text{iff } t^{-1} \text{ is a variable of sort } \mathsf{A} \\
chk(t, \mathbb{S}) & \text{iff } t \text{ is a variable of sort } \mathsf{S} \\
chk(t, \mathbb{D}) & \text{iff } t \text{ is a variable of sort } \mathsf{D} \\
chk(t, \mathbb{C}) & \text{iff } t \text{ is a variable of sort } \mathsf{C}
\end{array} \tag{11}
$$

$$
\frac{E(v_1) = E(v_2) \quad enc\_free\ E(v_1)}{as(E, c, u, \ulcorner v_1 \approx v_2 \urcorner, E, c, u)} \tag{12}
$$

$$
\frac{E(v_1) = E(v_2)^{-1} \quad enc\_free\ E(v_1)}{as(E, c, u, \ulcorner \mathsf{invp}(v_1, v_2) \urcorner, E, c, u)} \tag{13}
$$

$$
\frac{E(v_1) = h \quad E(v_2) = t}{as(E, +[h, t] :: c, u, \ulcorner \mathsf{send}(v_1, v_2) \urcorner, E, c, u)} \tag{14}
$$

$$
as*(E, \langle\rangle, \langle\rangle, \langle\rangle, E) \tag{15}
$$

$$
\frac{as(E_1, c_1, u_1, x, E_2, c_2, u_2) \quad as*(E_2, c_2, u_2, s, E_3)}{as*(E_1, c_1, u_1, x :: s, E_3)} \tag{16}
$$

Fig. 3. Abstract execution statement semantics

At runtime, a program variable is associated with an element of a message algebra. This association is represented by an environment $E : V \rightharpoonup \widehat{\mathsf{Alg}_Y}$, a finite partial function. The semantics of a sequence of statements is specified using the relation $asret(E, c, u, s, o)$, where E is an environment, c is a trace in $(\pm\overline{\mathsf{Alg}_Y})^*$, u is a sequence of fresh terms in Alg_Y^*, s is a sequence of statements, and o is a sequence of outputs in Alg_Y^*.

$$
\frac{as*(E, c, u, s, E') \qquad E' \circ \langle v_0, v_1, \ldots\rangle = \langle t_0, t_1, \ldots\rangle}{asret(E, c, u, s \frown \langle \ulcorner \mathsf{return}(v_0, v_1, \ldots) \urcorner\rangle, \langle t_0, t_1, \ldots\rangle)} \tag{17}
$$

The semantics of the remaining statements are given in Fig. 3. The semantics of expressions are given in Fig. 2. Note that Eq. 7, 12, and 13 make assertions that some terms must be free of encryptions. The purpose of these restrictions has to

Statement	Trace	Fresh	Environment
initial	$\langle +[h, \{\!\|n\|\!\}_k], -[h, n]\rangle$	$\langle n\rangle$	$E_0 = \emptyset[v_0 \mapsto h][v_1 \mapsto k]$
$v_2 : \mathbb{D} \leftarrow \mathsf{frsh}$	$\langle +[h, \{\!\|n\|\!\}_k], -[h, n]\rangle$	$\langle\rangle$	$E_1 = E_0[v_2 \mapsto n]$
$v_3 : \mathbb{M} \leftarrow \mathsf{encr}(v_2, v_1)$	$\langle +[h, \{\!\|n\|\!\}_k], -[h, n]\rangle$	$\langle\rangle$	$E_2 = E_1[v_3 \mapsto \{\!\|n\|\!\}_k]$
$\mathsf{send}(v_0, v_3)$	$\langle -[h, n]\rangle$	$\langle\rangle$	$E_3 = E_2$
$v_4 : \mathbb{D} \leftarrow \mathsf{recv}(v_0)$	$\langle\rangle$	$\langle\rangle$	$E_4 = E_3[v_4 \mapsto n]$
$v_2 \approx v_4$	$\langle\rangle$	$\langle\rangle$	$E_5 = E_4$
$\mathsf{return}(v_2)$	$\langle\rangle$	$\langle\rangle$	$E_6 = E_5$

Fig. 4. Initiator procedure execution

do with the correct handling of probabilistic encryption and will be explained in Sect. 7.

The intuition behind the semantics can be gleaned from the statement semantics *as* in Fig. 3. Think of an environment, trace, fresh values triple (E, c, u) as a state, and a statement as a label. Figure 3 specifies a labeled transition system. It defines how the states evolve during the course of an execution. For a sameness test $\ulcorner v_1 \approx v_2 \urcorner$ (Eq. 12), the state does not change. Execution halts if the test fails. For a send statement $\ulcorner \mathsf{send}(v_1, v_2) \urcorner$ (Eq. 14), only the trace is updated. For a bind statement $\ulcorner v : k \leftarrow x \urcorner$ (Eq. 10), all three components of the state are updated as determined by the expression semantics *ae*. The trace is changed only in response to a $\ulcorner \mathsf{recv}(v_1) \urcorner$ expression (Eq. 8), and a fresh value is consumed only in response to a $\ulcorner \mathsf{frsh} \urcorner$ expression (Eq. 9). Sequences of state transitions are tied together in the natural way by $as*$ (Eqs. 15 and 16). The *asret* predicate (Eq. 17) ensures that the final statement in a procedure is a return statement, and that the outputs of the procedure are correctly retrieved from the final environment.

Definition 4 (Procedure execution). *Let* $p = \langle (v_0, k_0), \ldots, (v_{n-1}, k_{n-1})\rangle$ *and* $i = \langle i_0, \ldots, i_{n-1}\rangle$. *Execution* $e = \mathsf{e}_Y(c, i, o, u)$ *is an execution of procedure* $x = \mathsf{x}(p, s)$, *written* $exec(x, e)$, *iff*

1. *for all* $j < n$, $chk(i_j, k_j)$, *and*
2. $asret(E, c, u, s, o)$, *where* $E = \emptyset[v_0 \mapsto i_0] \cdots [v_{n-1} \mapsto i_{n-1}]$.

See Eq. 11 for the definition of chk.

Roletran generates the following procedures for the Unilateral Protocol.

Example 5 (Unilateral Protocol Procedures)

$$
\begin{aligned}
\mathsf{initp} = \mathsf{x}(\langle (v_0, \mathbb{C}), (v_1, \mathbb{A})\rangle,\ & \\
& v_2 : \mathbb{D} \leftarrow \mathsf{frsh} \\
& v_3 : \mathbb{M} \leftarrow \mathsf{encr}(v_2, v_1) \\
& \mathsf{send}(v_0, v_3) \\
& v_4 : \mathbb{D} \leftarrow \mathsf{recv}(v_0) \\
& v_2 \approx v_4 \\
& \mathsf{return}(v_2))
\end{aligned}
\qquad
\begin{aligned}
\mathsf{respp} = \mathsf{x}(\langle (v_0, \mathbb{C}), (v_1, \mathbb{I})\rangle,\ & \\
& v_2 : \mathbb{M} \leftarrow \mathsf{recv}(v_0) \\
& v_3 : \mathbb{D} \leftarrow \mathsf{decr}(v_2, v_1) \\
& \mathsf{send}(v_0, v_3) \\
& \mathsf{return}(v_3))
\end{aligned}
$$

The execution $\mathsf{inite} = \mathsf{e}_{h:\mathsf{C},n:\mathsf{D},k:\mathsf{A}}(\langle +[h, \{\!|n|\!\}_k], -[h, n]\rangle, \langle h, k\rangle, \langle n\rangle, \langle n\rangle)$ is an execution of procedure initp. The state transitions caused by this execution of procedure initp are shown in Fig. 4.

4.1 Correctness

Definition 6 (Liveness). *Procedure x is* live *for role r, iff there exists an execution e such that*

1. *e is a run of r, and*
2. *e is an execution of x.*

Definition 7 (Safety). *Procedure x is* safe *for role r, iff when*

1. *e is an execution of x, then*
2. *e is a run of r.*

Definition 8 (Correctness). *Procedure x* correctly implements *role r, iff x is live and safe for r.*

The Coq scripts that come with Roletran automatically prove that the Unilateral Protocol procedures it generates correctly implement their respective roles.

Consider the case in which Roletran mistakenly omitted the sameness test ($v_2 \approx v_4$) in the initiator procedure. The Coq scripts would determine that $\mathsf{e}_{c:\mathsf{C},n,n':\mathsf{D},k:\mathsf{A}}(\langle +[h, \{\!|n|\!\}_k], -[h, n']\rangle, \langle h, k\rangle, \langle n\rangle, \langle n\rangle)$ is an execution of procedure initp', but note that this execution violates the safety condition. The safety condition ensures that runs of a collection of procedures that correctly implement the roles of a protocol achieve the security goals of the protocol.

5 A Runtime with Probabilistic Encryption

This section presents message algebras, called concrete message algebras, that are very similar to the ones used by the abstract execution semantics. The only difference is the way in which they model encryption. The signature used by the previous algebras has one operation for encryption, $\{\!|(\cdot)|\!\}_{(\cdot)}$ (See Fig. 1), which suggests that two encryptions are the same if the plaintext and the key used to construct them are the same. This is not true for implementations of encryption in actual use. Instead, some randomness is added to an encryption during its construction in such a way that knowledge of the randomness is not needed to recover the plaintext by someone in possession of the decryption key.

Sorts:	$\mathsf{M}, \mathsf{A}, \mathsf{S}, \mathsf{D}$		
Subsorts:	$\mathsf{A} < \mathsf{M}, \mathsf{S} < \mathsf{M}, \mathsf{D} < \mathsf{M}$		
Operations:	$(\cdot,\cdot)$	$: \mathsf{M} \times \mathsf{M} \to \mathsf{M}$	Pairing
	$\{\!\|\cdot\|\!\}^i_{(\cdot)}$	$: \mathsf{M} \times \mathsf{M} \to \mathsf{M}$	Encryption
	$\#$	$: \mathsf{M} \to \mathsf{M}$	Hash
	$(\cdot)^{-1}$	$: \mathsf{M} \to \mathsf{M}$	Key inverse
	$\tau_0, \tau_1, \ldots$	$: \mathsf{M}$	Tag constants
Equations:	$(x^{-1})^{-1} = x$ for $x : \mathsf{A}$; $x^{-1} = x$ otherwise		

Fig. 5. Concrete crypto algebra signature

Figure 5 shows the signature used for concrete algebras that model probabilistic encryption. This signature features a family of encryption operations, $\{\!|(\cdot)|\!\}^i_{(\cdot)}$, one for each natural number i. The natural number is meant to represent the randomness used while creating the encryption. In concrete algebras, two encryptions created with the same plaintext and key are equal only if they were created using the same random value.

The algebra of interest is the order-sorted quotient term algebra generated by a set of declarations Y. The message algebra CAlg_Y is the carrier set for sort M. The definitions of traces, roles, and executions, extend to concrete algebras in the obvious ways.

Definition 9 (Forgetful function). *Let $\mathcal{F} : \mathsf{CAlg}_Y \to \mathsf{Alg}_Y$ be the obvious function that forgets the randomness used to create encryptions.*

Lemma 10. *For $x \in \mathsf{Alg}_Y$, if x is encryption free (enc_free x), then there exists a unique $y \in \mathsf{Alg}_Y$ such that $\mathcal{F}(y) = x$.*

Proof. By induction on the structure of y.

The lemma used in proofs follows.

Lemma 11. *For $x, y \in \mathsf{CAlg}_Y$, if enc_free$(\mathcal{F}(x))$ and $\mathcal{F}(x) = \mathcal{F}(y)$, then $x = y$.*

6 Concrete Execution Semantics

The concrete execution semantics is analogous to the abstract execution semantics except that references to message algebras are replaced with references to concrete message algebras. There is one big exception. When executing an encr expression, there must be a source of randomness for use in creating an encryption. To provide a source of fresh basic values, the abstract execution semantics threads a sequence of values through state changes. In the concrete execution semantics, a sequence of natural numbers γ is also threaded through state changes and used to create encryptions.

$$\frac{E(v_1) = t_1 \qquad E(v_2) = t_2}{ce(E, c, u, \iota :: \gamma, \ulcorner \mathsf{encr}(v_1, v_2) \urcorner, \{\!|t_1|\!\}^{\iota}_{t_2}, c, u, \gamma)} \tag{18}$$

$$\frac{E(v_1) = t_1 \qquad E(v_2) = t_2}{ce(E, c, u, \langle\rangle, \ulcorner \mathsf{encr}(v_1, v_2) \urcorner, \{\|t_1\|\}^0_{t_2}, c, u, \langle\rangle)} \tag{19}$$

Equation 19 handles the case in which the source of randomness has been exhausted.

Other than the case for the encr expression, the definition of the concrete execution semantics follows that of the abstract execution semantics in the obvious ways.

Definition 12 (Concrete procedure execution) *Assume* $p = \langle (v_0, k_0), \ldots, (v_{n-1}, k_{n-1}) \rangle$ *and* $i = \langle i_0, \ldots, i_{n-1} \rangle$. *Execution* $e = \mathsf{e}_Y(c', i', o', u')$ *is a concrete execution of procedure* $x = \mathsf{x}(p, s)$ *with randomness* γ, *written* $cexec(x, e, \gamma)$, *iff*

1. *for all* $j < n$, $chk(\mathcal{F}(i_j), k_j)$;
2. $csret(E, c, u, \gamma, s, o)$, *where* $E = \emptyset[v_0 \mapsto i_0] \cdots [v_{n-1} \mapsto i_{n-1}]$;
3. c' *is the result of mapping* c *using* $\mathcal{F}$;
4. $i' = \mathcal{F} \circ i$;
5. $o' = \mathcal{F} \circ o$; *and*
6. $u' = \mathcal{F} \circ u$.

7 Relating Execution Semantics

The proofs of the theorems stated in this section were performed using Coq and the proof scripts are available in the distribution of CPSA [8].

Theorem 13 (Faithfulness). $cexec(x, e, \gamma)$ *implies* $exec(x, e)$.

The proof of faithfulness is tedious but straightforward. The forgetful function in Definition 9 is used to map items in the concrete semantics to items in the abstract semantics, and then the proofs go through as expected.

Theorem 14 (Adequacy). $exec(x, e)$ *implies* $cexec(x, e, \gamma)$.

The proof of adequacy is tricky. Where there is a sequence of state transitions in the abstract execution semantics, one must find a corresponding sequence in the concrete execution semantics. During both sequences, an event in the trace is consumed when a send statement or a receive expression is encountered. The case of a receive expression is the easy situation. The received term in the complex algebra can be any term as long as applying the forgetful function to it produces the received term in the abstract algebra. However, the case of a send statement is quite different. The transmitted term in the complex algebra must agree with what is in the environment associated with the send statement's message variable. And the term in the environment depends on the particular sequence of random values consumed up to this point in the execution. Engineering a proof that maintains this property is what makes the proof tricky.

The proof of adequacy makes demands on both the abstract and concrete execution semantics. The proof depends on the fact that the following terms must not contain an encryption,

– the key used during a decryption (see Eq. 7),
– the terms compared with a sameness test (see Eq. 12), and
– the terms compared with an inverse key predicate test (see Eq. 13).

The lack of encryptions allow the use of Lemma 11.

With these checks in place, the means we use to validate compiler input/output pairs correctly handles probabilistic encryption.

8 Epilogue

The development of the Roletran compiler is part of a project aimed at addressing the fact that there are systems built on aging software components with questionable security. An approach to protecting such systems is to isolate each component, and mediate communication between the components using trusted software that achieves desired security goals. Verified implementations of protocols is a key component to our approach. Members of this project include Ian D. Kretz and Dan J. Dougherty. The project is led by Joshua D. Guttman.

The project has developed a runtime system in Rust for code generated by Roletran, and several test protocols have been analyzed and then translated into running code, the simplest of which is the Unilateral Protocol. The project has another compiler that compiles protocols that make use of state. Future work might include the construction of a verified runtime system for Roletran generated code.

The addition of channels to CPSA was due to yet another successful collaboration between Joshua and the author.

Acknowledgement. Paul D. Rowe provided valuable comments that improved this paper.

References

1. The Coq proof assistant reference manual (2021). http://coq.inria.fr
2. Bhargavan, K., Corin, R., Deniélou, P., Fournet, C., Leifer, J.J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, 8–10 July 2009, pp. 124–140. IEEE Computer Society (2009). https://doi.org/10.1109/CSF.2009.26
3. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–207 (1983). https://doi.org/10.1109/TIT.1983.1056650
4. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. **105**(2), 217–273 (1992). https://citeseer.ist.psu.edu/goguen92ordersorted.html
5. Guttman, J.D., Herzog, J.C., Ramsdell, J.D., Sniffen, B.T.: Programming cryptographic protocols. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 116–145. Springer, Heidelberg (2005). https://doi.org/10.1007/11580850_8
6. Guttman, J.D., Wand, M.: VLISP: a verified implementation of scheme. Lisp Symbolic Comput. **8**, 5–32 (1995). https://doi.org/10.1007/BF01128406

7. Liskov, M.D., Rowe, P.D., Thayer, F.J.: Completeness of CPSA. Technical report, MTR110479, The MITRE Corporation (2011). https://www.mitre.org/publications/technical-papers/completeness-of-cpsa
8. Ramsdell, J.D., Guttman, J.D.: CPSA4: A cryptographic protocol shapes analyzer. The MITRE Corporation (2018). https://github.com/mitre/cpsaexp
9. Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: proving security protocols correct. J. Comput. Secur. **7**(1), 191–230 (1999). http://content.iospress.com/articles/journal-of-computer-security/jcs117