Programming Cryptographic Protocols[*]

Joshua D. Guttman Jonathan C. Herzog John D. Ramsdell Brian T. Sniffen

The MITRE Corporation
{guttman, jherzog, ramsdell, bsniffen}@mitre.org

Abstract

A programming language for cryptographic protocols eases design and implementation of application-specific protocols for tasks such as electronic commerce and distributed access control. The language provides a minimal expressiveness useful for defining new protocols.

We give the language a semantics via strand spaces, so that the designer can prove that a new protocol meets the security goals. This semantics also motivates a compilation strategy, yielding protocol implementations faithful to their verified behavior.

We also aim to clarify the relation between the abstract models used in protocol verification and the actual behavior of protocols as implemented.

1 Introduction

Cryptographic protocol analysis offers a number of highly informative techniques, e.g. [17, 8, 19]. In addition, work on protocol design [14, 18] holds out the hope of hand-crafted protocols for electronic commerce and cross-organization distributed applications. These protocols must be faithful to the trust relations among the participants, meaning their requirements for authentication and access control. If many protocols will be invented in the coming years, abstractly justified for specific tasks, how can they be implemented in a uniform, reliable way?

We describe here a cryptographic protocol programming language CPPL allowing a designer to express protocols at the Dolev-Yao level of abstraction. CPPL and its semantics are motivated by the strand space theory [19], so that the designer can verify that a new protocol meets its confidentiality and authentication goals. Alternative semantics could be given by translating the domain-specific language into spi or the applied pi calculus [4, 3], allowing other verification methods [17, 1].

CPPL is intended to provide minimal expressiveness compatible with protocol design. First, a protocol run must respond to choices made by its peer, as encoded in different forms of message that could be received from the peer. Second, the principal on behalf of whom the protocol is executing must be able to dictate choices reflecting its trust management policy [2, 9, 20]. Finally, CPPL provides a mechanism to call subprotocols, so that design may be modularized. The interface to a subprotocol shows what data values must be supplied to it and what values will be returned back on successful termination. The interface also shows what properties the callee assumes about the input parameters, and what properties it will guarantee to its caller about values resulting from successful termination. These—branching on messages received, consulting a trust management theory, and subprotocols—are the three main forms of expressiveness offered by CPPL.

We also need some functionality from libraries. The libraries include a cryptographic library—used to format messages, to encrypt and decrypt, to sign and verify, and to hash—and a communications library. The latter connects to other principals on the network and manages network level channels to them. These channels need not achieve any authentication or confidentiality in themselves [15]. The third library is a trust management engine. The trust management engine allows us to integrate the protocol behavior with access control in a trust management logic [2, 6, 22], giving an open-ended way to control when to abort a run, and to control the choice between one subprotocol and another. Indeed, a primary goal of the paper is to present a way to define protocols motivated by the connection between trust management and protocols described in [20].

The language is organized around a specific view of protocol behavior. In this view, as a principal executes a single local run of a protocol, it builds up an *environment* that binds variables to values encountered. Some of these values are given by the caller as values of parameters when the protocol is initiated; some are chosen

[*] Supported by the MITRE-Sponsored Research program.

randomly; some are received as ingredients in incoming messages; and some are chosen to satisfy trust management requirements. These bindings are commitments, never to be updated; once a value has been bound to a variable, future occurrences of that variable (especially when expected in an incoming message) must match the value or else execution of this run aborts. The environment at the end of a run records everything learnt during execution. A selection of this information is returned to the caller.

Our treatment of trust management is tightly connected. We associate a formula with each message transmission or reception. The free variables of the formula are variables in the environment. The formula associated with a message transmission is a *guarantee* that the sender must assert in order to transmit the message. The formula associated with a message reception is an *assumption* that the recipient is allowed to *rely* on. It says that some other principal has previously guaranteed something. A protocol is *sound* if in every execution, whenever one principal P relies on P' having said a formula ϕ, then there was previously an event at which P' transmitted a message, and the guarantee formula on that transmission implies ϕ.

The same idea shapes our treatment of subprotocols. A local message, sent by the calling protocol, starts a subprotocol run. Hence, the caller makes a guarantee that the callee can rely on. When the subprotocol run terminates normally, it sends a message back to its caller; the callee now makes a guarantee that the caller can rely on in the remainder of its run.

Related Work Despite the large amount of work on protocol *analysis*, the predominant method for *designing* and *implementing* a new protocol currently consists of a prolonged period of discussion among experts, accompanied by careful hand-crafted implementations of successive draft versions of the protocol. The recent reworking of the IP Security Protocols including the Internet Key Exchange was an example, involving a complex and important cluster of protocols.

Languages for cryptographic protocols, including spi calculus and its derivatives [4, 3, 13, 10], have been primarily considered tools for analysis rather than as programming languages for implementation.

There has been limited work on compilation for cryptographic protocols, with [24, 23, 12] as relevant examples. We add a more rigorous model of protocol behavior, centered around the environment mentioned above. We provide clear interfaces to communications services and the cryptographic library. We stress a model for the choices made by principals, depending on a trust management interpretation of protocols and on an explicit pattern-matching treatment of message reception. A semantics ties our input language to the strand space model, and motivates the structure of our compiler.

Main Contributions This paper makes four main contributions. First, we define a model of protocol behavior, encoded in a very small language CPPL. It embodies the three main forms of expressiveness mentioned above. Second, we provide a strand space semantics for CPPL programs (Section 4). The semantics yields a finite set of strands for each program, and each strand is of finite length (Proposition 4). However, an infinite set of executions are possible when these strands (instantiated with different data values) interact with each other and an active adversary.

Third, we view subprotocol call and return as local secure communications, which we augment the strand spaces to model. New theorems allow proving security goals about the new mechanism (Section 5, Propositions 5–7). Finally, we describe a compilation strategy motivated by the semantics (Section 6).

2 Access Control via a Protocol

As a very small example, suppose that a server B wishes to offer a collection of data, and intends to transmit a datum to a client A in encrypted form, assuming that A is authorized to receive it. The authorization decision may depend on different factors: In some cases there is a confidentiality policy intended to prevent the wrong principals from learning a secret. In other cases, the goal is to deliver commercially valuable information, e.g. information consisting of stock quotations, only to subscribers who pay for it. Indeed, we may suppose that there are two classes of service. Customers on the expensive contract get the best quality of information, while a cheaper service may provide information that is less accurate or less up-to-date. In the case of stock quotations, the lower class of service may provide quotations rounded to the nearest eighth of a point, rather than in exact thirty-seconds, or it may provide quotes delayed by fifteen minutes. The protocol (Figure 1) is a variant of the Needham-Schroeder-Lowe protocol.

Here, D is the name of the datum that A wishes to receive, and N_a is a nonce used to assure authentication and freshness for the value of D. B freshly generates the session key K to be used to protect the datum, and A proves that it has received K (and wants genuinely to receive D) by means of message 3. Finally in the last message, B delivers the current value V of datum D encrypted with K, associated with the tag val to indicate that high quality data is contained. If cheaper data is delivered instead, the tag $\mathsf{lo\_val}$ informs the client, which can report this to its caller. In either case, the server wants to ensure that the client's request is recent,

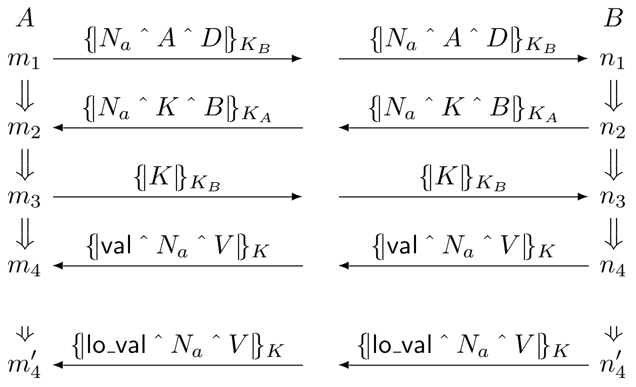
A
B
m_1
$\{\!|N_a \mathbin{\hat{}} A \mathbin{\hat{}} D|\!\}_{K_B}$
$\{\!|N_a \mathbin{\hat{}} A \mathbin{\hat{}} D|\!\}_{K_B}$
n_1
$\{\!|N_a \mathbin{\hat{}} K \mathbin{\hat{}} B|\!\}_{K_A}$
$\{\!|N_a \mathbin{\hat{}} K \mathbin{\hat{}} B|\!\}_{K_A}$
m_2
n_2
$\{\!|K|\!\}_{K_B}$
$\{\!|K|\!\}_{K_B}$
m_3
n_3
$\{\!|\mathsf{val} \mathbin{\hat{}} N_a \mathbin{\hat{}} V|\!\}_K$
$\{\!|\mathsf{val} \mathbin{\hat{}} N_a \mathbin{\hat{}} V|\!\}_K$
m_4
n_4
$\{\!|\mathsf{lo\_val} \mathbin{\hat{}} N_a \mathbin{\hat{}} V|\!\}_K$
$\{\!|\mathsf{lo\_val} \mathbin{\hat{}} N_a \mathbin{\hat{}} V|\!\}_K$
m_4'
n_4'

Figure 1: Access Control via Needham-Schroeder-Lowe

rather than being a replay, as preparing the data may cost time or money.

2.1 Growth of Environment during Execution

In a run of the client A, the environment must be initialized with values for A, B, D provided by the caller, indicating respectively the principal's identity, the server to interact with, and the data value to be retrieved. In addition, the environment must provide a value for K_B, B's public asymmetric encryption key. The first action of the client is to select a fresh random value N_a. With this, the environment contains values for all the ingredients in the first message to be sent. A call to a cryptographic library can cause a message of the correct form to be formatted as a bitstring. A call to a communication library can cause the bitstring to be sent, in the hope of the network delivering it to B.

When the communication library delivers a message, apparently from B, the cryptographic library attempts to decrypt it with A's private key K_A^{-1} (assuming the latter is available). If the first component of the resulting plaintext is not N_a, or if the last component is not B, then execution aborts. Otherwise, the middle component is bound to the variable K; it will be used as a symmetric key. The client can now format the third message with a call to the cryptographic library, and send it with a call to the communication library.

Finally, when the client receives the last message, the cryptographic library can decrypt it with the key bound to variable K. If the first component is the tag val, then the second component is bound to the variable V; another variable can be let-bound to indicate that high precision data was received. If instead it contains tag $\mathsf{lo\_val}$, V should still be bound, and the auxiliary variable should take a different value.

A symmetric sequence of actions occurs when the server executes a run.

$\mathsf{pubkey}(A, K_A)$	A's public encryp. key is K_A
$\mathsf{requests}(A, B, D, N)$	A requests D's value from B using N
$\mathsf{supply}(D, N)$	D's value to be supplied via N
$\mathsf{curr\_val}(D, V, N)$	D's value is V via nonce N
$\mathsf{approx\_val}(D, V, N)$	D is near V via nonce N

Table 1: Trust Management Predicates for Example

2.2 Trust Management Interpretation

We annotate the protocol by attaching trust management formulas to the nodes [20]. The formulas for our example, using the predicates shown in Table 1, are shown in Table 2. We regard node m_1 as asserting A's desire to receive the value of D from B. B ascertains that this event has occurred only at node n_3. In the meantime, B asserts at node n_2 that if A desires to receive D's value from B, then B will supply it. After n_3, B can infer that A has asserted its desire to receive D's value, which B is now committed to transmitting, either in the high-precision form or the low-precision form. To decide which, the server attempts to prove that A is entitled to high precision data; if it succeeds, it takes the branch containing val and high precision data. Otherwise, it takes the alternate branch with low precision data. The guarantees on n_4 and n_4' assert that the current value of D is V to the selected precision.

Each principal works within its own local theory to infer an instance of a *guarantee* formula before transmitting the message the formula guards. The local theory is augmented with the *rely* formulas associated with messages previously received. These assumptions make sense because the sender P previously guaranteed some formula ϕ. Thus, the assumption takes the form $P \mathsf{\ says\ } \phi$. The recipient P' can decide whether to trust P to speak truly about the matter ϕ, depending on what P' knows about P (i.e. formulas included in P''s local theory) and the contents of ϕ. P' *trusts* P on a subject ϕ if P' accepts the implication $(P \mathsf{\ says\ } \phi) \supset \phi$.

A protocol is *sound* if in every execution the formulas ρ_n on which principals rely are always true, in the sense that an uncompromised principal made the assertion that ρ_n says it made (Definition 9).

Different techniques could be used by principals wanting to determine how to proceed in a run of this protocol via logical inference and an explicitly represented theory. Frequently Datalog, or some variant or extension, will be used [22, 7]. Since γ_{n_2} contains an implication, an additional trick is needed here, namely checking that the conclusion is deducible when the hypothesis is added to the theory temporarily.

γ_{m_1}	$\mathsf{requests}(A, B, D, N_a)$
γ_{n_2}	$\mathsf{pubkey}(A, K_A)$ and $\mathsf{requests}(A, B, D, N_a) \supset \mathsf{supply}(D, N_a)$
ρ_{n_3}	A says $\mathsf{requests}(A, B, D, N_a)$
γ_{n_4}	$\mathsf{curr\_val}(D, V, N_a)$
$\gamma_{n'_4}$	$\mathsf{approx\_val}(D, V, N_a)$
ρ_{m_4}	B says $\mathsf{curr\_val}(D, V, N_a)$
$\rho_{m'_4}$	B says $\mathsf{approx\_val}(D, V, N_a)$

Table 2: Trust Management Annotations

```
statement  := RETURN
  |  LET id = expression IN statement
  |  <-- send_branches
  |  --> id recv_branches

empty :=              /* nothing */

send_branches := empty
  |  (GUARANTEE formula; SEND id msg;
      statement)
     send_branches

recv_branches := empty
  |  (RECEIVE msg; RELY formula;
      statement)
     recv_branches
```

Table 3: Syntax of Statements

3 Protocol Syntax

We now describe CPPL. Initially, we describe message transmission and reception, later adding mechanisms for declaring the interface to a protocol and for expressing subprotocol call and return.

3.1 Transmission and Reception

The main syntactic category of CPPL is the statement. A statement (Table 3) may return immediately, it may let-bind a variable to the value of an expression in a statement, or it may offer a choice of message transmissions (`<--`) or message receptions (`-->`).

Each message transmission branch is guarded by a formula. The branch may be selected only if the trust management engine succeeds in establishing ("guaranteeing") an instance of this formula. The environment for the remainder of the run is augmented with the instantiated variables. If the trust management engine fails on every branch's guarantee, then this run aborts.

Each message reception is guarded by a message pattern `msg`. That branch may be selected only if the parser recognizes an instance of this pattern in the message received from the communication layer, on the channel `id`. Variables not previously bound are instantiated to the values found by the parser. The trust management engine can rely on the formula associated with the selected pattern. The formula is added to the local theory as a temporary premise, so it can be used in proving guarantee formulas in the remainder of this run. If the message received from the communication layer does not match the pattern in any branch of a message reception statement, then this run aborts.

```
expression := id        |  constant
  |  NEW kind
  |  CHANNEL(id)        |  ACCEPT()
  |  REMOTE(id)

msg := id               |  constant
  |  msg, msg
  | {| msg |}_id        | [| msg |]_id
  | HASH(msg)
```

Table 4: Expressions and Message Patterns

Expressions are either identifiers, constants, *new* nonces or keys, or forms interacting with the communications library (Table 4). New nonces and keys are randomly generated by the cryptographic library. A *channel* expression requests a channel from the communications layer intended for bidirectional communication with the principal named by the argument; however, the channel provides neither authentication of origin nor confidentiality. The *accept* expression is used to act as a server; it yields a channel when a remote principal opens a connection. The *remote* expression queries the communication layer for the unauthenticated remote endpoint of the channel.

Message patterns interact with the cryptographic library instead of the communication layer. They allow identifiers and constants to be specified as atomic message patterns; concatenation of message patterns is indicated by the comma; and encryption and signature are indicated by `{| msg |}_K` and `[| msg |]_K` respectively. Hashing is also represented.

The syntax of formulas is not described here; it varies depending on the trust management engines that principals will use. The parser delegates these formulas to a parser specific to the trust management engine.

The AC NSL Example We specify the server (B) behavior of our sample protocol in Table 5. It starts by let-binding a channel and the principal at the remote end of it. A message is received off the channel which must match the pattern, with A appearing as the middle

```
let Chan = accept() in
let A    = remote(Chan) in
  --> Chan
   (receive {| N_a, A, D |}_KB;
    rely true;
    let K = new key in
    <--
     (guarantee gamma_n2;
      send Chan {| N_a, K, B |}_KA;
      --> Chan
      (receive {| K |}_KB
       rely rho_n3;
       <--
        (guarantee gamma_n4;
         send Chan {| val, V |}_K;
         let C = "hi_cost" in
          return)
        (guarantee gamma_n4';
         send Chan {| lo_val, V |}_K;
         let C = "lo_cost" in
          return))))
```

Table 5: Server Behavior in AC NSL

component. As a consequence of the message reception, N_a and D are bound to the values matched in the message. No assumption is delivered in the rely statement, as a message of this form could have been prepared by anyone, even an attacker knowing B's public key. B then binds a newly generated value to serve as the session key K, and asks the trust management engine to guarantee `pubkey(A,KA)` as part of γ_{n_2}. Although A was bound by a let-form, K_A is as yet unbound. Thus, the trust management engine operates in a logic programming style, delivering an extension to the environment in which K_A is bound to some key for which the trust management engine proved `pubkey(A,KA)`.

On the server side, the decision before the last transmission is a trust management decision. The trust management engine is asked to guarantee that the client A deserves high value information about datum D. If this fails, then the server tries the next branch.

On the client side, the branch is on the form of message received, specifically which tag is embedded (Table 6). On the two branches, the variable C is bound to different constants, and the protocol can return this value to its caller, together with the value V, so that the latter will be correctly interpreted.

```
--> Chan
  (receive {| val, V |}_K;
   rely B says curr_val(D,V,N_a);
   let C = "hi_cost" in
    return)
  (receive {| lo_val, V |}_K;
   rely B says approx_val(D,V,N_a);
   let C = "lo_cost" in
    return)
```

Table 6: Client Choice in AC NSL

```
procedure   := id (params) RELY formula
                : (params) GUARANTEE formula
              = statement end

statement  :=     ...     (see Table 3)
 |  <-> subprot_call

call_site  := invocation ELSE invocation

invocation := (GUARANTEE formula;
               id(params): params;
               RELY formula;
               statement)
```

Table 7: Syntax for Protocols and Call Sites

3.2 Subprotocols

Since one wants to construct protocols by using others as subprotocols, each protocol has an interface, and can use other protocols according to their stated interfaces (as in Table 7). The interface allows values to be passed to the subprotocol by its caller. The interface also specifies which parameters are to be returned by the subprotocol if it completes successfully. The subprotocol returns no values to its caller if it aborts.

The interface also includes two formulas. One is a formula serving as a precondition. Its free variables should be only the input parameters to the subprotocol, and it expresses a relationship among their values that the subprotocol designer assumes to hold. The caller must assure that this relationship holds before calling the subprotocol. The other formula concerns the values returned by the subprotocol. It expresses a relationship that the subprotocol will guarantee in all cases of successful termination. It must contain free only the input and output parameters of the protocol. The procedure headers for AC NSL are shown in Table 8.

The syntax for a subprotocol call site mirrors this structure. It contains two branches, each guarded by a guarantee formula. The branch will not be taken unless the trust management engine ensures an instance of the guarantee. The call site names a subprotocol to which it passes actual parameters. Values returned by the subprotocol may be required to match known values, while others will be bound to variables for use in the remainder of the caller's execution. The call site

```
ac_nsl_serv (B,KB,KBpriv)
    rely      pubkey(B,KB)
          and key_pair(KB,KBpriv)
 : (A,C,D,V)
    guarantee supplied(A,C,D,V) = ...

ac_nsl_client (A,KA,KApriv,B,D)
    rely      pubkey(A,KA) and
              key_pair(KA,KApriv)
: (N_a,C,V)   guarantee
          (C = "hi_cost" and
           B says curr_val(D,V,N_a))
       or (C = "lo_cost" and
           B says approx_val(D,V,N_a))
  = ...
```

Table 8: Procedure Headers for AC NSL

specifies a *rely* formula the summarizing the effect of the subprotocol, which the trust management engine can use in the remainder of the caller's execution.

Mismatch between values returned by the call and expected values, or an abort by the subprotocol, means that this branch has failed, and that the next branch will be tried. If both branches fail, the caller aborts.

A Certificate Retrieval Subprotocol In Table 5, we assumed that the server's trust management theory could supply a value K_A such that pubkey(A, K_A). Although some clients' public keys are known, others' must be retrieved from a directory of certificates. A subprotocol can be used to retrieve them, ensuring that they are sufficiently fresh. Although there are many strategies for doing so, there is a single criterion for the task, namely that the local trust management theory learns the conclusion pubkey(A, K_A). The rely-guarantee framework uses the subprotocol to ensure this follows from the local theory together with rely statements that become available. The correctness of these rely statements follows from protocol soundness as given in Definition 9.

We show in Table 9 a protocol to retrieve a public key from a directory D. The directory serves principal-public key bindings signed by a certificate authority C that may be verified using its (well-known) signature verification key C_{ver}. D ensures freshness using some certificate revocation mechanism, so D is trusted not to serve any revoked certificate. The assumption directory_service(D, C) makes explicit this trust, which is reflected logically as an implication allowing B to infer C says pubkey(A, K_A) from D says C says pubkey(A, K_A). When certifying_authority(C, A), C says pubkey(A, K_A) implies pubkey(A, K_A).

```
retrieve_pubkey
 (B,A,C,Cver,D,KD)
  rely certifying_authority(C,A) and
       sign_verification_key_of(C,Cver) and
       directory_service(D,C) and
       pubkey(D,KD)
 : (A,KA) guarantee pubkey(A,KA) =
 let Chan = channel(D) in
 let N_b = new nonce in
 let K = new key in
 <--
 (guarantee true;
  send Chan {| certify_key, A, K, N_b |}_KD;
  --> Chan
  (receive  {| cert_delivery, N_b,
                 [[ cert, A, KA ]]_Cver |}_K;
   rely D says C says pubkey(A,KA);
   return)) end
```

Table 9: Certificate Retrieval Protocol

B may try first to retrieve K_A locally if it is known to the local trust management theory, and only if this fails invoke the certificate retrieval protocol (Table 10).

The directory server runs a procedure matching retrieve_pubkey, in which D guarantees C says pubkey(A, K_A) before sending the certificate.

3.3 Complete Programs

A complete program is a set of procedures, each named by a different identifier. Each subprotocol call appearing in these procedures must be the name of a procedure also in the set. There is no constraint on the order in which the procedures are defined, nor on whether they call each other recursively. Any individual execution of a protocol will involve only a finite number of principals, executing the procedures a finite number of times (not necessarily to completion), and engaging in a finite number of transmissions and receptions.

4 Strands as a Semantics

In this section we first (Sections 4.1–4.2) resume the strand space theory. Section 4.3 provides a semantics for our protocol language by associating a protocol in the sense of Definition 3 to each program in the language.

The semantics requires a new ingredient in the strand space framework, to model the assumption of security for local message delivery such as subprotocol call and return. The same addition also allows reasoning about transport mechanisms that ensure confidential message delivery or authentication of message ori-

```
null_protocol () rely true
            : () guarantee true =
  return end

retrieve_pubkey_if_needed
  (B,A)    rely         true
: (A,KA)  guarantee    pubkey(A,KA) =
  <->
  (guarantee pubkey(A,KA);
   null_protocol():();
   rely true;
   return)
  else
  (guarantee certifying_authority(C,A)
         and sign_verification_key_of(C,Cver)
         and directory_service(D,C)
         and pubkey(D,KD);
   retrieve_pubkey(B,A,C,Cver,D,KD):(A,KA);
   rely D says C says pubkey(A,KA);
   return) end
```

Table 10: Certificate Retrieval when Needed

gin [11]. In Section 5 we explain how to prove security properties in this augmented framework.

4.1 Terms and Messages

Terms form a free algebra, built from atomic terms via constructors. The atomic terms are partitioned into the types *principals*, *texts*, *tags*, *keys*, and *nonces*, which are used for normal messages, together with *activation identifiers* which are used to represent subprotocol call and return messages, but never appear in real messages handled by the communication layer. In the present formulation, *channels* never occur within messages, although useful extensions could include that.

Some atoms are regarded as indeterminates (variables), while others are regarded as constants, representing particular data values used in a concrete run of a protocol. However, no tags are used as variables; instead, they are always protocol-specific constants found literally in messages, such as val and lo_val in the AC NSL protocol example. We will continue to write tags in sans serif font, while writing atoms a in general in italics.

The terms in the algebra A are freely built up from atoms using the operations of *concatenation*, *encryption*, *signature*, and *hashing*. These are written $t_0 \,\hat{}\, t_1$, $\{\!|t|\!\}_K$, $[\![\, t \,]\!]_K$, and $\mathsf{hash}(t)$ respectively. In the present formulation the second argument to an encryption or signature is always an atomic key. Our convention for public key encryption and digital signature is that the key K is always the public key. In a public key encryption $\{\!|t|\!\}_K$, K is the public encryption key, and in a digital signature $[\![\, t \,]\!]_K$, K is the public verification key.

A *substitution* is a finite function α mapping atoms to atoms, such that (1) α respects types in the sense that $\alpha(a)$ is an atom of the same type as a, and (2) the domain of α consists only of variable atoms. We insist that the range of a substitution is included in the atoms, because the theory is less attractive when substitutions may map variables to compound terms.

The application of a substitution α to a term t, written $t{\cdot}\alpha$, is defined as expected: application of α to terms is the homomorphism extending α's action on atoms.

4.2 Strands, Protocols, and Bundles

Definition 1 *A* direction *is one of the four symbols* $+, -, +_c, -_a$. *A* directed term *is a pair* $\langle d, t\rangle$ *with* $t \in \mathsf{A}$ *and* d *a direction. We write directed terms* $+t$, $+_c\, t$, *etc.* $(\pm\mathsf{A})^*$ *is the set of finite sequences of directed terms.* $\langle d, t\rangle \cdot \alpha = \langle d, t \cdot \alpha\rangle$.

A strand space *over* A *is a set* Σ *with a trace mapping* $\mathsf{tr} : \Sigma \to (\pm\mathsf{A})^*$ *and a substitution application operator* $s \cdot \alpha$ *such that*

$$\mathsf{tr}(s \cdot \alpha)(i) = (\mathsf{tr}(s)(i)) \cdot \alpha$$

for all $s \in \Sigma$, α, *and* i *such that* $1 \leq i \leq \mathsf{length}(s)$.

Here we regard $(\pm\mathsf{A})^*$ as the set of functions from initial sequences of positive integers to directed terms.

Message transmission has positive direction $+, +_c$, and reception has a negative direction $-, -_a$. The strands we construct in Section 4.3 to give semantics to a CPPL program are sequences of pairs, each consisting of a directed term and a formula; in this case the function tr is the function $\mathsf{map\ first}$ that returns the sequence of first elements. Some additional definitions, including the subterm relation $\sqsubset$ and the penetrator strands, are in Appendix A. Strands that are not penetrator behaviors are called *regular strands.*

Transmission that preserves confidentiality is a special kind of message transmission; a node of this kind we annotate with a subscript c on the positive sign. For instance, $+_c\, t$ means transmission of t via some method assumed to preserve confidentiality. Dually, reception that provides authenticity is a special kind of message reception indicated by a subscript a as in $-_a\, t$. If a communication arrow $n \to n'$ ensures both confidentiality and authentication, then n has annotation $+_c\, t$ and n' has annotation $-_a\, t$. Purely local communication such as subprotocol call or return is of this kind.

The set $\mathcal{N}$ of all nodes forms a directed graph $\langle \mathcal{N}, (\to \cup \Rightarrow)\rangle$ together with both sets of edges $n_1 \to n_2$ for communication and $n_1 \Rightarrow n_2$ for succession on the same strand (Definition 10). A *bundle* is a subgraph of $\langle \mathcal{N}, (\rightarrow \cup \Rightarrow)\rangle$ for which the edges are causally well-founded, expressing a possible execution. The content of the annotations $+_c, -_a$ comes from a modified notion of bundle, in which the transmission $+_c\ t$ is delivered only to a regular node, not a penetrator node, and in which a reception $-_a\ t$ arrived from a regular node.

Definition 2 *Let $\mathcal{B} = \langle \mathcal{N}_\mathcal{B}, (\rightarrow_\mathcal{B} \cup \Rightarrow_\mathcal{B})\rangle$ be a finite acyclic subgraph of $\langle \mathcal{N}, (\rightarrow \cup \Rightarrow)\rangle$. $\mathcal{B}$ is a* bundle *if:*

1. *If $n_2 \in \mathcal{N}_\mathcal{B}$ and term(n_2) is negative, then there is a unique n_1 such that $n_1 \rightarrow_\mathcal{B} n_2$.*
2. *If $n_2 \in \mathcal{N}_\mathcal{B}$ and $n_1 \Rightarrow n_2$ then $n_1 \Rightarrow_\mathcal{B} n_2$.*

$\mathcal{B}$ is a bundle with secure communication*, or* sc-bundle*, if in addition:*

3 If $+_c\ t \rightarrow_\mathcal{B} n$ or $n \rightarrow_\mathcal{B} -_a\ t$, then n is regular.

A notion of assured delivery could also be added to this framework; this would be a property $+_d\ t$ such that if $n_0 = +_d\ t \in \mathcal{N}_\mathcal{B}$, then there exists an $n_1 \in \mathcal{N}_\mathcal{B}$ such that $n_0 \rightarrow_\mathcal{B} n_1$.

In Section 5 we describe how to prove security-relevant conclusions about sc-bundles.

We assume given some logic L, by which we mean a set of formulas $FORMULA_L$ with a notion of substitution and a consequence relation $\Delta \longrightarrow \phi$ [16]. The formulas of L express trust management assertions.

Since we regard some atoms (other than tags) as variables, we consider a strand to be parameterized by the variable atoms in it. We sometimes write a strand s in the form $s[\vec{x}]$ to indicate that the variables appearing in it make up the list $\vec{x}$. Its instances are the strands obtained from it by applying a substitution α that may replace the atoms in $\vec{x}$ with others of the same types. As in [20], we define:

Definition 3 (Protocol) *An* annotated protocol Π *consists of a set of regular strands $\{s_j\}_{j\in J}$ together with a pair of functions γ and ρ from nodes of these strands to formulas of L, such that γ is defined on positive nodes and ρ is defined on negative nodes. The strands s_j are the* roles *of the protocol.*

The strand space Σ_Π over Π *consists of all instances of the parametric strands $s_j[\vec{x}]$ together with all penetrator strands from Definition 11.*

Substituting constants $\vec{c}$ for the variables $\vec{x}$ provides the same value for all occurrences of a variable in s_j, matching our operational view, that a variable once bound retains the same value throughout an execution.

Idea for a Semantics of Subprotocol Call Confidential transmission and authenticated reception suggest a semantics for subprotocol call and return. The call and return will be successive nodes ($+_c$ and $-_a$ respectively) on the strand s representing the behavior of the caller. The activity of the callee is represented on a separate regular strand s'.

The first node of s' is a negative $-_a$ node that accepts the actual parameters with which s' starts; the last node is a $+_c$ node that returns values to the caller. Further subprotocol calls from s' are executed on other strands s''. This is akin to a remote procedure call semantics, if one views the parts of the activity executing within different procedure invocations as making up separate strands.

We also include a uniquely originating value a[1] in the invocation and return messages. This *activation identifier* a abstracts the stack frame of the call. The call is a transmission node $n_0 = +_c\, t$ on s where $a \sqsubset t$, and the return is a reception node $n_1 = -_a\, t'$ where likewise $a \sqsubset t'$. Proposition 6 is the reason for including activation identifiers in the semantics; it tells us how to show the existence of a subprotocol strand in the same sc-bundle, that is, a regular strand s' which receives t and eventually transmits t'. To ensure that s' is the expected subprotocol, we stipulate that t has the form

$$\mathsf{call} \,\hat{}\, \mathsf{prot\_name} \,\hat{}\, p \,\hat{}\, a \,\hat{}\, \vec{b}.$$

Here call is a special tag indicating that this is a subprotocol call message, prot_name is a tag, which names the procedure defining the callee, p is the identity of the principal on behalf of which both caller and callee are executing, a is the activation identifier, and the $\vec{b}$ are the arguments passed as actual values to be bound to the formal parameters of the callee. We stipulate that the return term t' has the form

$$\mathsf{return} \,\hat{}\, \mathsf{prot\_name} \,\hat{}\, p \,\hat{}\, a \,\hat{}\, \vec{c}.$$

Here return is a special tag indicating that this is a subprotocol return message, and $\vec{c}$ are the actual values being returned to the caller.

Because subprotocol call uses a message transmitted from caller to callee, there should be a guarantee formula guarding the call site, and a rely formula at the beginning of the subprotocol. Because subprotocol return uses a message transmitted from callee to caller, the end of the subprotocol should assert a guarantee, and a rely formula should follow the call site. Hence the syntax given in Table 7.

4.3 The Protocol associated with a Program

To give a strand space semantics to a program in CPPL, we would like to assign to every program a protocol in

[1] See Definition 10; a uniquely originating value in a particular bundle is one that was created only at one node.

the sense of Definition 3, i.e. a set of parametric strands $\{s_j[\vec{x}]\}_{j\in J}$, together with rely and guarantee formulas for all of the negative and positive nodes (respectively) on these strands. The strand space Σ_Π describing all local runs of protocol procedures is thus the set of instances of the roles s_j, obtained by applying all substitutions to them, together with the penetrator strands of Definition 11.

The program consists of a number of procedure definitions of the form shown in Table 7, all having different names. We in fact define the semantics independently for each procedure. For each procedure we define a set of parametric strands, and the semantics of the whole program is simply the union of these sets. The heart of the semantics is a process that defines a set of strands recursively on the definition of statements.

In the strand space Σ we will construct, the strands s are sequences of pairs we call "events"; each event consists of a directed term and a formula. Thus, a strand s is in $(\pm\mathsf{A} \times \mathit{FORMULA}_L)^*$; i.e. it is a partial function from an initial sequence of positive integers to events in $\pm\mathsf{A} \times \mathit{FORMULA}_L$.

We write $x :: s$ to mean the partial function f on positive integers such that $f(1) = x$ and $f(i+1) = s(i)$; this is defined on an initial sequence if f is. We use nil for the trace without events, i.e. the everywhere undefined function. We write $s \frown s'$ for the result of appending s' after s, i.e. the function f such that $f(i) = s(i)$ if $i \le k = \mathsf{length}(s)$, and $f(i) = s'(i-k)$ otherwise. When S, T are sets of sequences, $S \frown T = \{s \frown t\colon s \in S \text{ and } t \in T\}$.

Suppose that we are giving the semantics for a procedure with name name, which will be used by principal p, will be activated with an activation identifier ai, and we let the return parameters for the procedure be the vector of atoms $\vec{b}$. We write $\vec{q} = \mathsf{name} \mathbin{\hat{}} p \mathbin{\hat{}} ai$ for the concatenation of this call information. For brevity, we write $\vec{v}$ for $\vec{q}, \vec{b}, \gamma_{ret}$, so that $t_{ret}(\vec{v}) = \mathsf{return} \mathbin{\hat{}} \vec{q} \mathbin{\hat{}} \vec{b}$ is the expected return message. If the guarantee formula associated with this normal termination of this procedure is γ_{ret}, we let

$$ret(\vec{v}) = (+_c\, t_{ret}(ai),\ \gamma_{ret})$$

be the return event expected from this procedure, consisting of message and guarantee.

We also define the abort event from the procedure to be the event

$$abort(ai) = (+_c\, \mathsf{abort} \mathbin{\hat{}} \vec{q},\ \mathit{true})$$

since no non-trivial formula results from failure.

If $s \in (\pm\mathsf{A} \times \mathit{FORMULA}_L)^*$ is a strand, then $\mathit{AIS}(s)$ is the set of activation identifiers that occur in the events of s. Since activation identifiers do not occur in formulas, this is the same as the activation identifiers occurring in messages sent or received in s.

$$\begin{aligned}
\mathcal{S}_{\vec{v}}(\texttt{<--}\ \beta_1 \ldots \beta_k) &= \bigcup_{1\le i\le k} \mathcal{S}_{\vec{v}}(\beta_i)\\
\mathcal{S}_{\vec{v}}(\beta_i) &= \{\langle +t_i, \gamma_i\rangle :: s\colon s \in \mathcal{S}_{\vec{v}}(\sigma_i)\}\\
\mathcal{S}_{\vec{v}}(\texttt{-->}\ c\ \beta'_1 \ldots \beta'_k) &= \bigcup_{1\le i\le k} \mathcal{S}_{\vec{v}}(\beta'_i)\\
\mathcal{S}_{\vec{v}}(\beta'_i) &= \{\langle -t_i, \rho_i\rangle :: s\colon s \in \mathcal{S}_{\vec{v}}(\sigma_i)\}
\end{aligned}$$

Table 11: Semantics of Send and Receive Statements

We give the semantics of statements, within a given procedure, by a semantic function $\mathcal{S}_{\vec{q},\vec{b},\gamma_{ret}}$, where the three parameters make available the call information $\vec{q}$, the return parameters $\vec{b}$, and the return guarantee γ_{ret} of the procedure. The statement semantics $\mathcal{S}_{\vec{v}}(\sigma)$, where σ is a statement, returns a set of strands. Every behavior σ can engage in is an instance of some strand in the set $\mathcal{S}_{\vec{v}}(\sigma)$.

Semantics of Return The behavior of a return call has one form, in which only the return message for the parameters $\vec{v} = \vec{q}, \vec{b}, \gamma_{ret}$ is transmitted:

$$\mathcal{S}_{\vec{v}}(\texttt{return}) = \{ret(\vec{v}) :: \mathsf{nil}\}$$

Semantics of Sending and Receiving A send branch conses an event—consisting of the sent message paired with the guarantee guarding the send—to the front of any behavior of the following statement. If a send statement has a number of branches, the semantics is non-deterministic, taking the union of the behaviors possible for the send branches, together with an abort if all branches are refused. The semantics of a receive statement is similar, but with the opposite sign.

This is summarized in Table 11, where we assume the statement consists of k branches, of which the ith send branch β_i takes the form `(guarantee` γ_i; `send` $c_i\ t_i$; σ_i`)`. The ith receive branch β'_i takes the form `(receive` t_i; `rely` ρ_i; σ_i`)`. Channels are discarded in the semantics, since the Dolev-Yao adversary controls the network and misroutes messages as desired.

Semantics of Subprotocol Call The semantics of procedure call involves the events that may occur when a subprotocol is tried, even though it does not successfully commit. A branch commits if the invoking transmission is followed by a successful return message.

If a branch's guarantee fails and there is no invocation, it has not committed. If an invocation receives no reply, either because the caller times out or because

$$
\begin{array}{rcl}
\mathcal{U}_s(b_i) & = & \{\mathsf{nil}\} \\
 & \cup & \{(+_c\,\mathsf{call}\,\hat{}\,\mathsf{name}\,\hat{}\,p\,\hat{}\,ai'\,\hat{}\,\vec{b_i}, \gamma_i) :: \mathsf{nil}\} \\
 & \cup & \{(+_c\,\mathsf{call}\,\hat{}\,\mathsf{name}\,\hat{}\,p\,\hat{}\,ai'\,\hat{}\,\vec{b_i}, \gamma_i) :: \\
 & & \quad (-_a\,\mathsf{abort}\,\hat{}\,\mathsf{name}\,\hat{}\,p\,\hat{}\,ai', \mathit{true}) :: \mathsf{nil}\} \\
\mathcal{C}_s(b_1) & = & \{(+_c\,\mathsf{call}\,\hat{}\,\mathsf{name}\,\hat{}\,p\,\hat{}\,ai'\,\hat{}\,\vec{b_i}, \gamma_i) :: \\
 & & \quad (-_a\,\mathsf{return}\,\hat{}\,\mathsf{name}\,\hat{}\,p\,\hat{}\,ai'\,\hat{}\,\vec{b_i}, \rho_i) :: \mathsf{nil}\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_{\vec{v}}(b_1\ \texttt{else}\ b_2) & = & \bigcup_{s_2 \in \mathcal{S}_{\vec{v}}(b_2)} \mathcal{U}_{s_2}(b_1) \frown \{s_2\} \\
 & \cup & \bigcup_{s_1 \in \mathcal{S}_{\vec{v}}(b_1)} \mathcal{C}_{s_1}(b_1) \frown \{s_1\} \\
 & \cup & \mathcal{U}_{\mathsf{nil}}(b_1) \frown \mathcal{U}_{\mathsf{nil}}(b_2) \frown \{\mathit{abort}(ai_0) :: \mathsf{nil}\}
\end{array}
$$

Table 12: Uncommitted and Committed Behaviors, choosing variable $ai' \notin (\mathit{AIS}(s) \cup \{ai_0\})$, and Subprotocol Call

the callee tries to return values that do not match those already bound in the caller, then there is no commit. Finally, if the guarantee succeeds but the invocation causes an abort, then it has not committed.

Thus, the *uncommitted behavior* of a subprotocol call branch is a strand of length 0, 1, or 2. When a subprotocol call site executes its else branch, some uncommitted behavior for the main branch will be prepended to the behavior of the else branch. If the else branch also fails to commit, then the caller must abort. The *committed behavior* of a successful subprotocol is a strand of length 2, namely a call and a successful return, prepended to the behavior of the statement it contains.

Subprotocol invocation semantics is in Table 12, where the call site is β_1 `else` β_2, and each β_i is:

$$\texttt{guarantee}\ \gamma_i;\ \mathsf{name}\ \vec{b_i} : \vec{c_i}\ \texttt{rely}\ \rho_i;\ \sigma_i.$$

Here $\vec{b}, \vec{c}$ are the call and return parameters; and γ_i and ρ_i are the formulas to guarantee and to rely on. Let p be the current principal, and ai_0 the activation identifier of the strand executing this call. $\mathcal{U}_s(b)$ is the set of uncommitted behaviors b may contribute preceding the behavior of the strand s, while $\mathcal{C}_s$ is the set of committed behaviors b may contribute preceding the behavior of the strand s.

The activation identifier used on a subprotocol invocation and return is chosen to be distinct from the activation identifiers $\mathit{AIS}(s)$ used later on the same strand s, and distinct from the activation identifier ai_0 received by that strand when it was called. This is the only role of the parameter s. The choice of ai' is made in some canonical way from all other activation identifiers.

$$
\begin{array}{rcl}
\mathcal{S}_{\vec{v}}(\texttt{let}\ i = i'\ \texttt{in}\ \sigma) & = & \mathcal{S}_{\vec{q}, \vec{b}[i'/i], \gamma_{ret}[i'/i]}(\sigma[i'/i]) \\
\mathcal{S}_{\vec{v}}(\texttt{let}\ i = e\ \texttt{in}\ \sigma) & = & \mathcal{S}_{\vec{v}}(\sigma) \\
 & & (e \text{ not of form } i')
\end{array}
$$

Table 13: Semantics of *let* Statements

Semantics of *let* Statements We divide *let* statements `let` $i = e$ `in` σ into two kinds (Table 13). Either the expression e is an identifier i, or else it is a *new*, *remote*, or channel expression. When the expression e is an identifier i', we interpret the *let* statement by substituting i' in place of the target i throughout $\mathcal{S}_{\vec{v}}(\sigma)$.

In the second case, when e is not an identifier, we simply ignore the *let*: the semantics of the whole statement is identical with the semantics of σ. When e is a channel expression, this is natural, as channels are invisible at the level of the Dolev-Yao semantics, and in fact i will not appear in the semantics of σ. When e is a *remote* expression, it evaluates (at run time) to some principal that the communication layer believes to be at the far end of the channel. However, the resulting strands are viewed as parametric roles. Thus, the role will have instances in which any principal is substituted for i. This is just the desired interpretation, as any principal could be at the far end of a new channel.

4.4 Properties of this Semantics

This semantics is in one sense a *finite* representation of the behavior of CPPL programs. Suppose that `proc_name` is a procedure where the nesting depth of parentheses introduced by `<--`, `-->`, and `<->` statements is d. Suppose the branching factor at each such statement is at most k. That is, each `<--` or `-->` statement has no more than k branches, and if there are any `<->` statements in the procedure, then $k \geq 2$. Suppose that the statement forming the body of `proc_name` is σ.

Proposition 4 *The cardinality* $|\mathcal{S}_{\vec{v}}(\sigma)| \leq k^d$. *If* $s \in \mathcal{S}_{\vec{v}}(\sigma)$, *then* $\mathsf{length}(s) \leq (4 \cdot d) + 1$.

The maximum value $(4 \cdot d) + 1$ is attained when s has a sequence of subprotocol calls, and each involves a call and an abort on its main branch followed by a call and a return to commit to its alternative branch; at the very end is one `return`.

If a CPPL program contains j procedures each satisfying these bounds, then the resulting protocol Π contains at most $j \cdot k^d$ roles. Typically, k will be quite small, d will be relatively small, and in fact $|\mathcal{S}_{\vec{v}}(\sigma)|$ will be much less than k^d because many statements embedded within σ will have fewer than k branches. For instance, the server procedure in the version of AC NSL

(Table 5) satisfies $k = 2$ and $d = 4$. However, the semantics generates only two roles, one for each choice in the innermost transmission statement.

In a different sense, the behaviors generated by a protocol may be infinite, because there are infinitely many sc-bundles $\mathcal{B}$ such where the regular strands are instances of the roles of this protocol. To reason about this infinite collection of bundles, we need some general theorems, which we present next.

5 Reasoning about Secure Communication

To reason about authenticated or confidential communication, we use analogues to the unsolicited and outgoing test principles (Propositions 15, 14). The principle for authenticated communication is simple:

Proposition 5 *If $\mathcal{B}$ is an sc-bundle and n is a regular node $n \in \mathcal{B}$ with sign $-_a$, then there exists a regular node $m \in \mathcal{B}$ such that $m \to n$, hence $term(m) = term(n)$.*

For confidential communication, we use a simpler analogue to the outgoing test:

Proposition 6 *Suppose that a originates uniquely within $\mathcal{B}$ (an sc-bundle) at node n_0 with direction $+_c$, and $a \sqsubset term(n_1)$. Suppose there is no positive node n' such that $a \sqsubset term(n')$ and $n_0 \Rightarrow^* n' \prec n_1$. Then in $\mathcal{B}$ there exists a regular edge $m_0 \Rightarrow^+ m_1$ such that $n_0 \to m_0 \Rightarrow^+ m_1 \preceq_{\mathcal{B}} n_1$.*

When n_0 and n_1 lie on the same strand, then the ordering relation $n_0 \prec m_0 \prec m_1 \preceq n_1$ establishes the recency of m_0, m_1. Time may be measured in a purely local way along $n_0 \Rightarrow^+ n_1$, allowing the communication layer of the implementation to abort execution by raising a timeout before the n_1 could occur. Thus, m_0 cannot have occurred longer ago than the locally chosen timeout value.

When Proposition 6 is used for reasoning about subprotocol call, $n_0 \Rightarrow n_1$, and a node n' cannot exist.

5.1 Tail Recursive Subprotocol Call

The tail call optimization can be validly performed with a small change to this semantics. When making an optimized tail call, the subprotocol passes the same activation identifier ai that it was given. This expresses the idea that the new call overwrites and reuses the top stack frame. The only alteration to Table 12 is that in this case $ai' = ai_0$, and the original caller should not insist that the return or abort contains the same procedure name name; instead, a different value name′ is accepted.

The principle needed to reason about subprotocol calls with the tail call optimization is designed to be usable several times in succession. It is therefore somewhat more complex than Proposition 6.

To state the principle, we need an auxiliary definition. Given an sc-bundle $\mathcal{B}$ with ordering $\preceq_{\mathcal{B}}$, we say that a set N of nodes is *a, n_1-full* if (1) a originates uniquely in $\mathcal{B}$ on some node $n_0 \in N$; (2) whenever $a \sqsubset \mathrm{term}(n)$ and there exists an $n' \in N$ such that either (2a) $n \preceq_{\mathcal{B}} n'$ or else (2b) $n' \Rightarrow^+ n \prec_{\mathcal{B}} n_1$, then it follows that also $n \in N$.

Proposition 7 *Let $\mathcal{B}$ be a bundle with $n_1 \in \mathcal{B}$ and $a \sqsubset term(n_1)$. Let N be a, n_1-full and suppose that every positive node in N is $+_c$, and $term(n_1) \neq term(n)$ for every $n \in N$.*

There exist regular nodes $m_0, m_1 \in \mathcal{B}$ such that $m_0, m_1 \notin N$ and $n \to m_0 \Rightarrow^+ m_1 \preceq n_1$ for some $n \in N$.

Proof. Let a originate uniquely at n_0 by (1), and $S =$

$$\{m \in \mathcal{B} \colon m \notin N \text{ and } a \sqsubset \mathrm{term}(m) \text{ and } m \preceq_{\mathcal{B}} n_1\}.$$

S is non-empty as $n_1 \in S$, so S has minimal elements (Proposition 13). Let m_0 be a minimal element of S. By (2b), m_0 does not follow any member of N on the same strand. By (2a), m_0 does not precede any member of N on the same strand. Since by [25, Lemma 2.9] there is a sequence of arrows leading from n_0 to m_0, and as we have just seen, the last arrow is not $\Rightarrow$, it must end with an arrow $n \to m_0$. Since $n \in N$ is positive, n is $+_c$ and m_0 is regular. Moreover, m_0 is negative and $\mathrm{term}(n) = \mathrm{term}(m_0)$. Thus, $\mathrm{term}(m_0) \neq \mathrm{term}(n_1)$ and $m_0 \neq n_1$, hence there is a non-empty chain of arrows leading from m_0 to n_1. As m_0 is negative, the first arrow of this chain must be $m_0 \Rightarrow m_1$. ▮

In using Proposition 7 to reason about tail call, n_0 represents the original call site where the activation identifier a originates, and n_1 represents the ultimate return, so that $n_0 \Rightarrow n_1$. The set N represents a set of calls to new strands; condition (2a) says that N is connected in the sense that it contains all intermediate strands lying between n_0 and later calls in N with activation identifier a. In the case of tail call, condition (2b) tells us that when N contains any node of a strand s', then we should add to N all the nodes up to the final tail call (which re-uses the activation identifier a). The assumption that $n \in N$ implies $\mathrm{term}(n) \neq \mathrm{term}(n_1)$ says that no regular strand already in N executes a return that could match n_1. Then the proposition tells us to infer that there is another regular strand containing $m_0 \Rightarrow^+ m_1$ which is another tail call using a.

We developed the theory of sc-bundles in this paper to provide a semantics for subprotocol call. However, it has independent interest, allowing us to replicate within strand spaces the theory of "security transactions using secure transport protocols" of Broadfoot and Lowe [11]. In establishing that protocols are correct, assuming that certain messages pass over a medium ensuring confidentiality or authenticity, one would use Propositions 5 and 15, as well as an additional result combining the content of Propositions 7 and 14.

5.2 Protocol Soundness

Two definitions from [20] help clarify the relationship between the notion of bundle and the trust management annotations that allow a principal to control its protocol executions. *Permissibility* formalizes the idea that a guarantee formula must be proved by a principal, using its local theory and earlier rely formulas as premises.

Definition 8 *A regular strand s of Σ_Π is* permissible *for Π and Th up to k if, for each $i \leq k$ such that $s \downarrow i = n$ is positive, γ_n is derivable from $\{\rho_m : m = s \downarrow j$ is negative and $j < i\}$ in Th.*

Suppose each principal P holds theory Th_P. An sc-bundle $\mathcal{B}$ is permissible *if every regular strand s is permissible for Th_P up to its $\mathcal{B}$-height k.*

Permissible bundles are the only ones that can really happen, assuming that all of the *regular* principals play by the rules and do not execute n without proving γ_n from earlier ρ_ms. The adversary does not have to prove anything, and adversary nodes have no annotations.

Central to our approach is the notion of *protocol soundness.* A protocol is sound if, in every execution, when one principal engages in a negative node n, then ρ_n is a consequence of the assertions other principals made on earlier nodes m. Naturally, soundness can assume that the execution respects some unique origination and non-origination assumptions, satisfied in a set $\mathbb{B}$. The principal executing node m is called $\mathsf{prin}(m)$.

Definition 9 Soundness. *Bundle $\mathcal{B}$* supports *a negative node $n \in \mathcal{B}$* relative to theory Th *iff ρ_n is a consequence of the set of formulas $\{\mathsf{prin}(m)\,\mathsf{says}\,\gamma_m : m \prec_\mathcal{B} n\}$ in Th. If Π is an annotated protocol, and $\mathbb{B}$ is a set of sc-bundles, then* Π is sound for $\mathbb{B}$ *in Th if, whenever $\mathcal{B} \in \mathbb{B}$, for every negative $n \in \mathcal{B}$, $\mathcal{B}$ supports n.*

Soundness results are essentially a form of authentication theorem. They say that every bundle containing a node n contains certain earlier, "supporting" nodes. For instance, the proof that a bundle of AC NSL supports the server's third node n_3 is very similar to standard proofs of the responder's guarantee in NSL (see for instance [19]). It uses the same unique origination and non-origination assumptions. It would be false were this protocol based on the original Needham-Schroeder protocol, which is an unsound basis for the trust management goals of AC NSL.

A sound protocol is a coordination mechanism. Principals reason purely locally, using their own theories. The *rely* formulas they use as premises are however coordinated with guarantees derived by other principals, when the protocol is sound.

6 Compilation

The compiler for CPPL[2] is organized around a runtime environment, which records the values bound to atoms. These values are bitstrings and other implementation-level objects such as communications channels. Each execution step modifies this runtime environment. For instance, a *let* statement augments the environment with a new binding. A *receive* statement causes the received message to be parsed; the right bitstrings must be recognized for atoms that are already bound, and other values that were encountered will be installed, bound to atoms that were not previously bound. A *send* statement can also cause atoms to be bound, because the trust management system, operating in a logic-programming style, delivers new bindings that make the guarantee formula true.

Each statement is compiled to a procedure of one argument, namely the runtime environment. Each such procedure may do communication or trust management reasoning before selecting which statement to continue with. It executes a tail call to the procedure compiled from that statement, with a possibly extended runtime environment.

Compiler state The compiler maintains two main data structures as it makes a recursive descent through the abstract syntax tree of a statement. One is a compile-time environment, which maps atoms to indices into the runtime environment. The runtime environment is implemented simply as a vector.

The other data structure is a list of the *rely* formulas that have been traversed on the descent, starting with the *rely* formula given in the procedure interface. On traversing each receive statement branch, the compiler augments this list with the newly encountered *rely* formula. On traversing each branch of a transmit statement, the compiler generates a call to the trust management engine that packages these *rely* formulas as extra premises, available in proving an instance of the current *guarantee* formula. In this call, the runtime environment will also be passed as a parameter, so that the trust management engine—if successful—can return the extended runtime environment.

[2] As of this writing (July 15, 2004), the compiler is not yet implemented. Parts of a prototype have been written.

```
let rec iter_send rte = function
    []               -> abort ()
  | (gamma, chan, fmt, resume) :: rest ->
      match TME.infer rte rhos gamma with
        None         -> iter_send rte rest
      | Some rte'    ->
          Comms.send (rte chan) (fmt rte);
          (resume rte')

let rec iter_recv msg rte = function
    []               -> abort ()
  | (parser, resume) :: rest   ->
      match (parser msg) with
        None         -> iter_recv msg rte rest
      | Some rte'    -> resume rte'
```

Table 14: Send and Receive Dispatching

Control The main runtime control is shown in Table 14 expressed in OCaml. The compiler considers the branches of send and receive statements in sequence, so its code is more deterministic than the semantics of Table 11. This refinement is unproblematic, as the security properties we wish to establish (e.g. authentication and protocol soundness) are safety properties.

A send branch consists of: a formula `gamma`; an index into the runtime environment pointing to the channel to send the message on; a procedure `fmt` to format the message using values from the runtime environment; and `resume`, the procedure representing the statement embedded within this branch. `TME.infer` invokes the trust management engine to prove an instance of `gamma` from the available *rely* formulas in the current run-time environment. To fail, it returns `None`. To succeed it returns a `Some rte'`, where `rte'` extends `rte`.

When a message `msg` is received from the communications layer, it is provided to the successive receive branches, each of which contains a parsing procedure and a resumption to apply to the extended environment if the parser succeeds. The parsing procedure, like the format procedure used in send branches, must be generated by the compiler.

Format Functions The compiler constructs the format function `fmt` for each transmission branch by a recursive descent through its message pattern, determining a sequence of calls to the cryptographic library.

Message Parsers Constructing the message parser for a receive branch is more challenging. Some keys may be delivered in the message, and then used to decrypt other parts of it, which may in turn furnish other keys. The order in which to process the parts may not be obvious. The *pattern* of a message, in the sense of Abadi-Rogaway [5], records what portions of the message are accessible. If the pattern of a message (starting with the current compile-time environment) is identical with the message, then the message can be fully decoded. Otherwise, we raise a compile-time error. Calculating the pattern of a message also indicates the order dependencies for destructuring parts of the message. The optimal parser respects these dependencies, so as to decode submessages containing keys, before encrypted submessages that use those keys.

7 Conclusion

We have described CPPL, a programming language with just the expressiveness needed to express cryptographic protocols at the Dolev-Yao level. CPPL's semantics supplies a finite set of parametric strands defining the possible behaviors of any program. The strand space theory can be used to prove security properties of programs, using existing techniques and theorems newly introduced here for reasoning about subprotocol call. A compilation strategy is suggested by the semantics, organized around a runtime environment. The compiler issues calls on a cryptographic library to parse messages at run time. It issues calls on a trust management engine to choose future behavior.

In future work we will incorporate primitives such as Diffie-Hellman; this should be possible in a cryptographically sound way [21]. Mechanized analysis of the strands generated from CPPL programs is under development. Also desirable would be to incorporate existing certificate parsers into the run-time message parsers, so certificates in standard formats, contained in the messages, can be recognized and their content made available to the trust management engine.

References

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just Fast Keying in the pi calculus. In David Schmidt, editor, *Programming Languages and Systems: ESOP 2004 Proceedings*, number 2986 in LNCS, pages 340–354. Springer Verlag, January 2004.

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–34, September 1993.

[3] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL '01)*, pages 104–115, January 2001.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[5] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[6] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[7] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *17th Computer Security Foundations Workshop*, pages 139–154, Asilomar, CA, June 2004. IEEE CS Press.

[8] Bruno Blanchet. An efficient protocol verifier based on Prolog rules. In *14th Computer Security Foundations Workshop*, pages 82–96. IEEE CS Press, June 2001.

[9] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Distributed trust management. In *Proceedings, 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1997.

[10] Michele Boreale. Symbolic trace analysis of cryptographic protocols. In *ICALP*, 2001.

[11] Philippa Broadfoot and Gavin Lowe. On distributed security transactions that use secure transport protocols. In *Proceedings, 16th Computer Security Foundations Workshop*, pages 63–73. IEEE CS Press, 2003.

[12] Federico Crazzolara and Giuseppe Milicia. Developing security protocols in χ-spaces. In *Proceedings, 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.

[13] Federico Crazzolara and Glynn Winskel. Composing strand spaces. In *Proceedings, Foundations of Software Technology and Theoretical Computer Science*, number 2556 in LNCS, pages 97–108, Kanpur, December 2002. Springer Verlag.

[14] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system for security protocols and its logical formalization. In Dennis Volpano, editor, *Proceedings, 16th Computer Security Foundations Workshop*, pages 109–125. IEEE CS Press, 2003.

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

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

[17] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proceedings, 15th Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2002.

[18] Joshua D. Guttman. Authentication tests and disjoint encryption: a method for security protocol design. *Journal of Computer Security*, 2004. Forthcoming.

[19] Joshua D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.

[20] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In David Schmidt, editor, *Programming Languages and Systems: 13th European Symposium on Programming*, number 2986 in LNCS, pages 325–339. Springer, 2004.

[21] Jonathan C. Herzog. The Diffie-Hellman key-agreement scheme in the strand-space model. In *16th Computer Security Foundations Workshop*, pages 234–247, Asilomar, CA, June 2003. IEEE CS Press.

[22] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings, 2002 IEEE Symposium on Security and Privacy*, pages 114–130. May, IEEE Computer Society Press, 2002.

[23] Jonathan Millen and Frederic Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[24] Adrian Perrig and Dawn Xiaodong Song. A first step toward the automatic generation of security protocols. In *Network and Distributed System Security Symposium*. Internet Society, February 2000.

[25] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

A Additional Strand Notions

Definition 10 *Fix a strand space Σ:*

1. *The subterm relation $\sqsubset$ is the smallest reflexive, transitive relation such that $t \sqsubset \{\!|g|\!\}_K$ if $t \sqsubset g$, and $t \sqsubset g \,\hat{}\, h$ if either $a \sqsubset g$ or $a \sqsubset h$.*

   (Hence, for $K \in \mathsf{K}$, we have $K \sqsubset \{\!|g|\!\}_K$ only if $K \sqsubset g$ already.)

2. *A* node *is a pair $\langle s, i\rangle$, with $s \in \Sigma$ and i an integer satisfying $1 \le i \le \mathsf{length}(tr(s))$. We often write $s \downarrow i$ for $\langle s, i\rangle$. The set of nodes is $\mathcal{N}$. The* directed term *of $s \downarrow i$ is $\mathsf{tr}(s)(i)$.*

3. *There is an edge $n_1 \to n_2$ iff $term(n_1) = +t$ or $+_c\ t$ and $term(n_2) = -t$ or $-_a\ t$ for $t \in \mathsf{A}$. $n_1 \Rightarrow n_2$ means $n_1 = s \downarrow i$ and $n_2 = s \downarrow i + 1 \in \mathcal{N}$.*

   $n_1 \Rightarrow^* n_2$ (respectively, $n_1 \Rightarrow^+ n_2$) means that $n_1 = s \downarrow i$ and $n_2 = s \downarrow j \in \mathcal{N}$ for some s and $j \ge i$ (respectively, $j > i$).

4. *Suppose I is a set of terms. The node $n \in \mathcal{N}$ is an* entry point *for I iff $term(n) = +t$ for some $t \in I$, and whenever $n' \Rightarrow^+ n$, $term(n') \notin I$. t*

originates *on* $n \in \mathcal{N}$ *iff* n *is an entry point for* $I = \{t' : t \sqsubset t'\}$.

5. *An term* t *is* uniquely originating *in* $S \subset \mathcal{N}$ *iff there is a unique* $n \in S$ *such that* t *originates on* n*, and* non-originating *if there is no such* $n \in S$.

If a term t originates uniquely in a suitable set of nodes, then it plays the role of a nonce or session key. If it is non-originating, it can serve as a long-term shared symmetric key or a private asymmetric key.

Definition 11 *A* penetrator strand *is a strand* s *such that* $\mathsf{tr}(s)$ *is one of the following:*

M_t:	$\langle +t\rangle$ *where* $t \in$*text*
K_K:	$\langle +K\rangle$ *where* $K \in \mathsf{K}_{\mathcal{P}}$
$\mathsf{C}_{g,h}$:	$\langle -g,\ -h,\ +g \,\hat{}\, h\rangle$
$\mathsf{S}_{g,h}$:	$\langle -g \,\hat{}\, h,\ +g,\ +h\rangle$
$\mathsf{E}_{h,K}$:	$\langle -K,\ -h,\ +\{\!|h|\!\}_K\rangle$
$\mathsf{D}_{h,K}$:	$\langle -K^{-1},\ -\{\!|h|\!\}_K,\ +h\rangle$
$\mathsf{V}_{h,K}$:	$\langle -[\![\, h\,]\!]_K,\ +h\rangle$
$\mathsf{A}_{h,K}$:	$\langle -K^{-1},\ -h,\ +[\![\, h\,]\!]_K\rangle$
H_h:	$\langle -h,\ +\mathsf{hash}(h)\rangle$

A node is a penetrator node *if it lies on a penetrator strand, and otherwise it is a* regular *node.*

Definition 12 *A node* $n \in \mathcal{B} = \langle \mathcal{N}_{\mathcal{B}}, \rightarrow_{\mathcal{B}} \cup \Rightarrow_{\mathcal{B}}\rangle$ *if* $n \in \mathcal{N}_{\mathcal{B}}$*. The* $\mathcal{B}$-height *of a strand* s *is the largest* i *such that* $\langle s, i\rangle \in \mathcal{B}$ *or 0 if there is none.* $\prec_{\mathcal{B}}$ *is the transitive closure of* $\rightarrow_{\mathcal{B}} \cup \Rightarrow_{\mathcal{B}}$*, and* $\preceq_{\mathcal{B}}$ *is its reflexive, transitive closure.*

Proposition 13 *If* $\mathcal{B}$ *is a bundle,* $\preceq_{\mathcal{B}}$ *is a partial order. Every non-empty subset of the nodes in* $\mathcal{B}$ *has* $\preceq_{\mathcal{B}}$*-minimal members.*

We reason about secrecy using the notion of safety [19]. We write safe for *safe* keys, i.e. keys that the penetrator can never learn or use [19]. Since long term shared keys and private asymmetric keys are never transmitted in reasonable protocols, these keys are safe unless compromised before execution of the protocol. Session keys are safe if transmitted only on $+_c$ nodes or nodes protected by keys K with $K^{-1} \in \mathsf{safe}$.

If we consider the abstract syntax tree of a term t, and $t_0 \sqsubset t$, then there is a branch leading from the root (labeled t) to some subtree labeled t_0. Moreover, by Definition 10 Clause 1, this branch does not traverse any key edge leading from a term $\{\!|h|\!\}_K$ to its key K.

When S is a set of terms, t_0 *occurs only within* S in t if, regarding t as an abstract syntax tree, every branch from the root to an occurrence of t_0 that avoids key edges traverses some occurrence of a $t_1 \in S$ before reaching t_0. It *occurs outside* S in t if $t_0 \sqsubset t$ but t_0 does not occur only within S in t. A term t_0 *occurs safely* in t if it occurs only within $S = \{\{\!|h|\!\}_K : K^{-1} \in \mathsf{safe}\}$ in t.

The Authentication Tests given here are in a simpler and stronger form than in [19].

Proposition 14 (Outgoing Authentication Test) *Suppose* $\mathcal{B}$ *is a bundle in which* a *originates uniquely at* n_0*;* a *occurs only within* S *in* $term(n_0)$ *and* a *occurs safely in* S*; and* $n_1 \in \mathcal{B}$ *is negative and* a *occurs outside* S *in* $term(n_1)$.

There are regular $m_0, m_1 \in \mathcal{B}$ *such that* $m_0 \Rightarrow^+ m_1$*, where* m_1 *is positive,* a *occurs only within* S *in* $term(m_0)$*, and* a *occurs outside* S *in* $term(m_1)$*. Moreover,* $n_0 \preceq m_0 \prec m_1 \prec n_1$.

Proposition 15 (Unsolicited, Incoming Tests) *Suppose* $n_1 \in \mathcal{B}$ *is negative,* $\{\!|h|\!\}_K \sqsubset term(n_1)$*, and* $K \in \mathsf{safe}$*. (Unsolicited test:) There exists a regular* $m_1 \prec n_1$ *such that* $\{\!|h|\!\}_K$ *originates at* m_1*. (Incoming test:) If in addition* $a \sqsubset h$ *originates uniquely on* $n_0 \neq m_1$*, then* $n_0 \prec m_0 \Rightarrow^+ m_1 \prec n_1$.

B Full Syntax of Protocol Language

```
procedure  := id (params) RELY formula :
                (params) GUARANTEE formula
              = statement end

statement  := RETURN
            | LET id = expression IN statement
            | <-- send_branches
            | --> id recv_branches
            | <-> call_site

send_branches := empty
            | (GUARANTEE formula; SEND id msg;
               statement) send_branches

recv_branches := empty
            | (RECEIVE msg; RELY formula;
               statement) recv_branches

call_site  := invocation ELSE invocation

invocation := (GUARANTEE formula;
               id(params): params;
               RELY formula;
               statement)

msg        := id                | msg, msg
            | {| msg |}_id      | [[ msg ]]_id
            | HASH(msg)

expression := id
            | new_expr
            | REMOTE(id)        | channel_expr

new_expr   := NEW KEY           | NEW NONCE
```

```
channel_expr := CHANNEL(id)   |    ACCEPT()

params     := empty           |  id more_ps

more_ps    := empty           |  , id more_ps

id         := token           |  token : type


type       := PRINCIPAL   |  TEXT
           |  KEY         |  NONCE
           |  TAG

empty      :=      /* empty */
```