Programming Cryptographic Protocols*

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

The MITRE Corporation

Abstract. Cryptographic protocols are useful for trust engineering in distributed transactions. Transactions require specific degrees of confidentiality and agreement between the principals engaging in it. Moreover, trust management assertions may be attached to protocol actions, constraining the behavior of a principal to be compatible with its own trust policy. We embody these ideas in a cryptographic protocol programming language CPPL at the Dolev-Yao level of abstraction. A strand space semantics for CPPL shaped our compiler development, and allows a protocol designer to prove that a protocol is sound.

1 Introduction

In this paper, we describe the core of a cryptographic protocol programming language, CPPL, a domain specific language for expressing cryptographic protocols. It matches the level of abstraction of the Dolev-Yao model [15], in the sense that the programmer regards the cryptographic primitives as black boxes, and concentrates on the structural aspects of the protocol. CPPL allows the programmer to control protocol actions using trust constraints [23], so that an action such as transmitting a message will occur only when the indicated trust constraint is satisfied. We offer a semantics for CPPL in the style of structured operational semantics; this semantics identifies a set of *strands* [34] as the meaning of a role in a protocol. The semantics is useful for two reasons. First, it suggests a method by which the programmer may prove that a protocol meets its security goals [21]. Second, it clarifies issues of scope and binding, and therefore assisted us in implementing a correct compiler.

Trust Engineering. A domain specific language for cryptographic protocols raises the question, however, why programmers need to create new protocols. Although there could be several answers to this, one specific answer motivated our work on CPPL. When a programmer must implement a transaction in a distributed application, CPPL allows him to engineer a protocol to achieve the specific authentication and confidentiality goals needed by this transaction. This process—the process of shaping a transaction so that it can reflect the trust goals of its participants—we call *trust engineering*.

* Supported by the MITRE-Sponsored Research program. Authors' addresses: guttman, jherzog, ramsdell, bsniffen@mitre.org.

Moreover, each participant must understand at exactly which step in the protocol they undertake a commitment, such as the commitment to pay for some goods. If a principal P makes several successive commitments in a protocol, then P should be able to decide before each of these steps whether it is willing to incur that commitment. If not, it may prefer to select some alternative, or it may need to abort the transaction. The content of the commitment will depend on the constituents of the messages in this execution, for instance the cost of the purchase or the principal to whom the money should be transferred.

Thus, it is not sufficient to have a few specific security protocols, such as TLS or SSH; instead, different combinations of confidentiality and agreement are required in different transactions. Although a transaction may be implemented using TLS or SSH as a lower level medium for confidentiality or entity authentication, a protocol design problem still persists, of ensuring the right degree of agreement and secrecy between the participants, and of identifying the trust and commitments required for each step in the protocol.

The protocol design problem is pervasive in electronic commerce, web services, and other aspects of distributed applications. CPPL is intended to express the core functionality that programmers will need, if they are to use cryptographic protocols as a central mechanism in trust engineering, and especially to connect trust management [25] and protocols [23].

An Example. Suppose that we would like to go into business, offering on-line stock quotes to a set of clients registered as customers. On a particular occasion, a client will request a collection of data D, possibly representing a market sector; we assume that the value D also contains a transaction identifier that the client can use to re-identify this request when billed. The client and server use a Needham-Schroeder-like protocol [32] to agree on a session key, and then the server delivers a real time stream of data containing stock quotes for sector D. In Figure 1, we see that the session key SK replaces the responder's nonce of the Needham-Schroeder protocol; we assume for now that each principal has the other's public encryption key. The server B wants to authenticate A to ensure

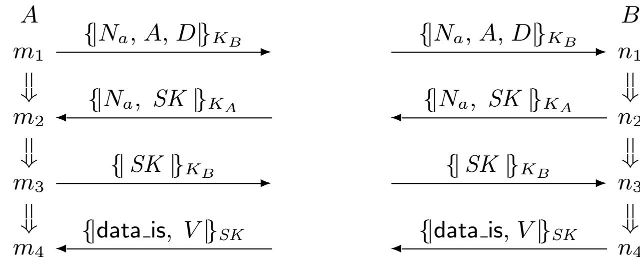

Fig. 1. NS Quote Protocol

that he can bill A for delivering this data. Conversely, the client A wants B to successfully authenticate its clients, so that A will not be charged for any service consumed by other clients C. A also needs to authenticate B, and ensure that the session key is shared only with B. This allows A to infer—based on a trust decision about B—that the data is accurate, timely, and therefore suitable for business use.

In this protocol, A is committing himself to the request for D in sending the message on node m_1. B learns that A has made this request when the authenticating handshake completes, which occurs when B receives the third message on node n_3. When sending the message on node n_4, B is committing itself to the assertion that V is an accurate stream of values for the market sector D. B also must guarantee to itself that A will pay for the data D before transmitting it; this decision may depend on databases of subscribers, accounts in arrears, and similar facts.

Structure of the paper. In Section 2, we summarize the main ideas of the language, describing its core syntax and informal semantics in Section 3. A strand-based semantics for individual local protocol runs in given in structured operational semantics format in Section 4, and the global execution semantics in terms of bundles is in Section 5. The strand space methods for proving results about protocols are adapted to this context in Section 6. Our stock quote service example is described in detail in Section 7.

2 Main Ideas of CPPL

CPPL is intended to provide only the minimal expressiveness necessary for protocol design, which calls for three fundamental ingredients. First, a protocol run must respond to choices made by its peer, as encoded in different formats 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, 7, 17, 23], using the choices to determine whether messages are sent, and if so of what format. Finally, CPPL provides a mechanism to encapsulate behaviors into 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 before transmission, and subprotocols—are the three main forms of expressiveness offered by CPPL.

We also rely on three libraries. The first is a cryptographic library, which is used to format messages, to encrypt and decrypt, to sign and verify, and to hash. The second is a communications library. It 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, 5, 28], giving an open-ended way to control when to abort a run, and to control the choice between one subprotocol and another. The trust management engine is free to determine the formulas expressing trust constraints. However, CPPL determines the set of values that may appear as individual constants in these formulas. These values are nonces, keys, and other values that we regard as texts; texts include addresses and names. The trust management engine maintains a theory, a set of formulas in the trust management logic. The theory is used to infer that trust constraints are satisfied; a theory may be augmented with new formulas as a protocol execution proceeds.

We associate a formula with each message transmission or reception. 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 as part of this protocol, and the guarantee formula on that transmission implies ϕ.

In the NS Quote protocol shown in Figure 1, on node m_1 the client guarantees that it is requesting the value of D from B. We represent this with the formula $\mathsf{requests}(A, B, D)$. At the end of the authentication phase, in node n_3, B has ascertains that this has occurred, and relies on the formula A says $\mathsf{requests}(A, B, D)$. Knowing that A has made this request presumably helps B be sure of being paid. On node n_4, B guarantees $\mathsf{will_pay}(A, D)$ and $\mathsf{curr_val}(D, V)$. The first part is intended to protect B itself, since B wants not to transmit the value V without an expectation of being paid. The second part is intended to protect A, that is, to ensure that A receives correct information. There is one other guarantee in this protocol. It guards node n_2, stipulating $\mathsf{owns}(A, K_A)$, i.e. that the value used to encrypt the second message is in fact the public key of A.

The same rely/guarantee idea shapes our treatment of subprotocols. A local message, sent by the calling protocol, starts a subprotocol run. Hence, the caller makes a guarantee on which the callee can rely. When the subprotocol run terminates normally, the callee sends a message back to its caller; the callee now makes a guarantee on which the caller can rely for the remainder of its run. Thus, a subprotocol call is a mechanism for the caller to discover the information guaranteed when the callee terminates successfully.

The Run-Time Environment. 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 identifiers to values encountered. Some of these values are given by the caller as values of parameters when the protocol is initiated; some are chosen randomly; some are received as ingredients in incoming messages; and some are chosen to satisfy trust manage-

ment requirements. These bindings are commitments, never to be updated; once a value has been bound to an identifier, future occurrences of that identifier must match the value or else execution of this run aborts. In particular, when a known value (such as SK) is expected in an incoming message (such as the message received on n_3), any other value will prevent execution of this run from continuing.

The environment at the end of a run records everything learnt during execution. A selection of this information is returned to the caller.

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 [24] for instance, involved a complex and important cluster of protocols.

Languages for cryptographic protocols, such as spi calculus [4, 3, 14, 8], have been primarily tools for analysis rather than programming languages.

There has been limited work on compilation for cryptographic protocols, with [33, 31, 13] 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 [21]. This semantics motivates the structure of our compiler; moreover, a designer can use it to verify that a new protocol meets its confidentiality and authentication goals. Alternatively one could translate CPPL into spi or the applied pi calculus [4, 3], allowing other verification methods [18, 1].

3 The CPPL Core Language

We describe here not the user-level syntax for CPPL, but a simplified syntax, which we call the CPPL *core language*. It provides information at the right locations to make the semantics easy to express, and likewise to direct the compiler. Users write programs in a different surface syntax, illustrated in Section 7.

The syntax of the CPPL core language is presented in Figure 2. The CPPL core language has procedure declarations and seven types of code statements. Programming language identifiers are indicated by x and y, and message tags by r. When used to concatenate message patterns, the comma operator is right associative, and tagging binds less tightly than comma. The language has syntax for guarantees and relies—by convention we write guarantees as Φ and relies as Ψ—which are finite lists of trust management formulas. We use finite lists, which we interpret conjunctively. Formulas in relies and guarantees may contain, in addition to logical variables and CPPL values, also CPPL identifiers. If bound in the environment at runtime, a CPPL identifier will be replaced in Φ, Ψ by the

$$
\begin{aligned}
p &\rightarrow \text{proc } p\ (x^*)\ \Psi\ c \\
c &\rightarrow \text{return } \Phi\ x^* \\
&\mid\ \ \text{let } x = \text{new in } c \\
&\mid\ \ \text{let } x = \text{accept in } c \\
&\mid\ \ \text{let } x = \text{channel } y \text{ in } c \\
&\mid\ \ (sb^*) \mid (x\ rb^*) \mid (cb^*) \\
sb &\rightarrow \text{send } \Phi\ x\ m\ c \\
rb &\rightarrow \text{recv } m\ \Psi\ c \\
cb &\rightarrow \text{call } \Phi\ p\ (x^*)\ (y^*)\ \Psi\ c \\
m &\rightarrow x \quad \mid \quad m, m' \quad \mid \quad r\, m \\
&\mid\ \ (m) \quad \mid \quad [\ m\ ]\ x \quad \mid \quad \{\ m\ \}\ x
\end{aligned}
$$

Fig. 2. CPPL Core Language

value to which it is bound; if not yet bound, it serves as a query variable that will be bound as a consequence of a trust management call. Logical variables in a trust management formula, if they occur, are interpreted implicitly universally.

A procedure declaration specifies the name p of the procedure, a list (x^*) of formal parameters, and a list of preconditions Ψ involving the formal parameters. The body of the procedure is a code statement c. A code statement may be: a return instruction, which specifies a list of postconditions Φ and return parameters (x^*); a let-statement; or a list of send branches, receive branches, or call branches. An identifier x is either a lowercase identifier `id`, or else an identifier with typing information `id:type`. We write $\mathsf{ide}(X)$ for the set of identifiers used in the phrase X.

A well-formed code statement c with two return statements at different locations must have the same postconditions Φ and return parameters x^*. Our translation from the user-level syntax to the core language ensures this.

NS Quote Example in CPPL. To illustrate the CPPL core language, we will return to the NS Quote example. We first focus on the protocol actions, leading to the behavior shown in Figure 3. We replace the trust management annotations with underscores to focus attention on the channels, new values, and messages. The

```
proc server (b:text, kb:key) _
  let chan = accept in
  (chan recv {na:nonce, a:text, d:text} kb _
    let sk:symkey = new in
    (send _ chan {na, sk, b} ka
      (chan recv {sk} kb _
        (send _ chan {Data_is v} sk
          return _))))
```

Fig. 3. The NS Quote Server's Behavior

server's parameters are its own name and public encryption key. It waits to accept an incoming connection, which the communication layer delivers as the bidirectional channel `chan`. It reads a message off this channel, which binds `na` to a nonce, and `a` and `d` to texts interpreted as a name and the desired data. The server generates a fresh session key `sk`, which is transmitted and received back in different encrypted forms to accomplish the authentication test of `a`'s identity. Finally, the current value is returned encrypted with the session key `sk`, tagged with Data_is to make its interpretation unambiguous.

We now insert the trust management information in italicized form in Figure 4. The procedure relies on the assumption that `kb` is really the public key that

```
proc server (b:text, kb:key) [owns(b, kb)]
  let chan = accept in
  (chan recv {na:nonce, a:text, d:text} kb [true]
    let sk:symkey = new in
    (send [owns(a, ka)] chan {na, sk, b} ka
      (chan recv {sk} kb [says_requests(a, a, b, d)]
        (send [will_pay(a, d); curr_val(d, na, v:text)]
              chan {Data_is v} sk
          return [supplied(a, na, d, v)]))))
```

Fig. 4. The NS Quote Server, with Trust Formulas

`b` owns, and states this assumption in its procedure header. The caller must arrange to start the server with values satisfying this assumption. The server learns nothing from the first message; it is encrypted using `b`'s public key, and could have been prepared by an adversary as well as a regular principal. The transmission of `sk` is guarded by a guarantee that `a` owns the public encryption key `ka`. We regard this as a query against a deductive database. As a consequence, either `ka` becomes bound to a suitable value, or the query fails, aborting execution of this run. Presumably, the server has a database of keys for all of its subscribers. After the next message is received, `b` has authenticated the peer `a`, and relies on `a` having said that `a` is requesting the data `d` from `b`. We use the predicate `says_requests(A, A', B, D)` to mean that A says `requests(A', B, D)`. This has the advantage of fitting the "says" locution into Datalog [10], our implementation's trust management logic, at least when only atomic formulas rather than compound formulas are said. It places a burden however on a principal—the server in this case—to include rules in its theory to allow `requests(A, B, D)` to be inferred from `says_requests(A, A', B, D)` for suitable values of the variables.

If `b` convinces itself that `a` will pay, and that the current value is `v`, then the value can be sent. The return parameters may be used by the caller for accounting and billing, with the guarantee that this data was supplied. We will extend the example in Section 7 to illustrate branching and subprotocols.

Informal execution semantics. To explain how procedures execute, we first introduce an auxiliary notion: *guaranteeing* formulas Φ in a runtime environment. This means to ask the runtime trust management system to attempt to ascertain the formulas Φ. Identifiers in Φ already bound in the runtime environment are instantiated to the associated values. Identifiers not yet bound in the runtime environment are instantiated by the trust management system, if possible, to values that make the formulas Φ true. The runtime environment extended with these new bindings is the result of successfully guaranteeing Φ. If the runtime trust management system fails to establish an instance of Φ the guarantee fails.

To execute a **return** statement, we attempt to guarantee the formulas Φ. If successful, we select from the resulting environment the values of each of the return parameters x^*; these values are returned to the caller. If the attempt to guarantee Φ fails, execution terminates abnormally, and the caller is informed of the failure. The caller receives no parameter values in case of failure.

To execute a list of **send branches**, the runtime trust management system selects a branch within which it can successfully guarantee the formulas Φ. The message pattern m specified on this branch, instantiated using the values in the resulting extended runtime environment, is then transmitted. Execution proceeds with the code c embedded within this send branch in the extended environment. If the runtime trust management system fails to guarantee the formulas Φ on any send branch, then execution terminates abnormally, and the caller is informed of the failure.

To execute a list of **receive branches** with identifier x, the runtime environment is consulted for the value bound to x. This value should be a channel. When a message is received over this channel, the message is matched against the patterns m within the receive branches. In a successful match, the message must agree with the runtime environment for identifiers in m that are already associated with a value. Other identifiers in m will be bound to the values observed in the incoming message, yielding an extended runtime environment. If at least one receive branch has a successful match, one such branch is selected. The formulas Ψ are instantiated using the extended runtime environment, and supplied to the runtime trust management system as additional premises. Execution proceeds with the code c embedded within this send branch in the extended environment. If no receive branch has a successful match, then execution terminates abnormally, and the caller is informed of the failure.

To execute a list of **call branches**, the system treats the call branches as sends followed by receives. That is, the the runtime trust management system selects a branch, within which it can successfully guarantee the formulas Φ. It calls the associated subprotocol procedure p with the parameters x^* instantiated using the values in the resulting extended runtime environment. This procedure may return normally, in which case it supplies values for the parameters y^*; execution continues with the embedded statement c, using the extended runtime environment. The instances of the formulas Ψ are supplied to the runtime trust management system as additional premises during execution of c. If p does not return normally, then execution may continue with a different call branch; ex-

ecution proceeds in the original environment, without any extension from the abnormally terminated call branch.

Local nature of this description. This execution semantics is local in the sense that it describes what one principal P does. This involves deciding what values to bind to identifiers; what messages to send; how to process a message that is received; and how to select a procedure to call as a subprotocol. It says nothing about how messages are routed on a network; nothing about what another principal P' does with messages received from P; nothing about how another principal P' created the messages that P receives. Likewise, it describes only the execution of one procedure. It says nothing about the behavior of a subprotocol invoked in a call branch. In essence, the execution semantics describes only a single principal executing a single run of a single procedure. Thus, it is natural to describe any single run by a strand. We describe how to do this in Section 4, and then describe what global executions are possible in Section 5.

4 Local Semantics

We give the semantics of CPPL procedures and code statements by describing the *strands* describing their possible behavior. Each strand specifies a sequence of transmissions and receptions that is possible for a principal executing this CPPL phrase faithfully.

Term Algebra. Each transmission or reception is a term in a free algebra A. The atomic terms are texts, nonces, and keys, denoted below as a. A compound term in A is either a concatenation $g,\ h$, a tagged message $\mathsf{tagname}\ g$, or the result of a cryptographic operation. In this section and the next, we will write the results of all cryptographic operations involving a plaintext g and an atomic key K in the form $\{|g|\}_K$. However, CPPL has syntax to distinguish symmetric and asymmetric operations, and to distinguish encryptions from signatures.

A direction is a value with polarity $+$ or $-$, which we use to indicate transmission and reception respectively. A directed term is a pair (d, t) where d is a direction and $t \in \mathsf{A}$.

Strand Spaces. A *strand space* Σ is a set equipped with a trace mapping tr such that $S \in \Sigma$ implies $\mathsf{tr}(S)$ is a finite sequence of directed terms. We regard finite sequences such as $\mathsf{tr}(S)$ as (1-based) finite partial functions defined on an initial segment of the positive integers. Σ is typically defined to be the union of a set of *regular* strands, representing the behaviors compatible with a protocol being studied, and a set of *penetrator* strands, representing behaviors within the capability of an adversary. Our standard adversary model is formalized in the Appendix as Definition 9.

Σ is an *annotated strand space* if in addition Σ is equipped with pair of functions γ, ρ, such that for all $S \in \Sigma$ and all positive integers i, if $\mathsf{tr}(S)(i)$ has positive [respectively, negative] direction, then $\gamma(S)(i)$ [respectively, $\rho(S)(i)$] is a

finite list of formulas. The formulas in the range of γ and ρ are called guarantee formulas and rely formulas respectively. We do not stipulate the logic to which the formulas belong, as the logic is an implementation-specific choice, which in our implementation is Datalog. The formulas are of interest only when $S \in \Sigma$ is regular; penetrator strands never make an enforceable commitment, and never rely on assertions of other principals. Thus, if S is a penetrator strand, then $\gamma(S)(i) = [\,]$ and $\rho(S)(i) = [\,]$ whenever they are defined.

Σ is a *strand space with uniqueness assumptions* if Σ is equipped with an operation Υ such that, for each $S \in \Sigma$, $\Upsilon(S)$ is a set of atoms that occur in $\mathsf{tr}(S)$; these are values that are uniquely originating in bundles of interest.

To give the semantics for a set of CPPL procedures, we define an annotated strand space with uniqueness assumptions. We give the semantics in the form of a Structured Operational Semantics. The primary judgments are of the form $\sigma; \Gamma \vdash \; c : s, \upsilon$. Here σ is a runtime environment, meaning a finite function mapping identifiers to values; Γ is a set of formulas serving as a theory; c is the code to be executed; and s, υ describes a strand. In this description, s describes the messages and associated formulated, while υ is a set of atoms containing the values assumed to have been freshly chosen. The judgment $\sigma; \Gamma \vdash \; c : s, \upsilon$ says that s, υ is one possible behavior that can result if c is executed in environment σ, when the principal holds theory Γ. A typical rule shows that a larger piece of code c_1 can unleash a strand of length $n+1$, assuming that a code statement c_0 embedded within c_1 can unleash a strand of length n. The behavior of c_0 describes everything after the first event of some behavior of c_1.

We describe strands $S \in \Sigma$ by grouping $\mathsf{tr}, \gamma, \rho$ together:

Definition 1 (Strand Descriptions). Let s be a finite sequence of pairs, where the first element in each pair is a directed term $\pm t$ and the second element in each pair is a list of formulas. A sequence of length 1 $\langle(\pm t\,,\ \Phi)\rangle$ *describes* a strand $S \in \Sigma$ iff the length of S is 1, $\mathsf{tr}(S)(1) = \pm t$; if its direction is $+$, then $\gamma(S)(1) = \Phi$; if its direction is $-$, then $\rho(S)(1) = \Phi$.

Sequence $(-t\,,\ \Psi) \Rightarrow s_0$ describes S if for some $S_0 \in \Sigma$, s_0 describes S_0 and

1. $\mathsf{tr}(S)(1) = -t$ and $\rho(S)(1) = \Psi$;
2. $\mathsf{tr}(S)(i+1) = \mathsf{tr}(S_0)(i)$, $\gamma(S)(i+1) = \gamma(S_0)(i)$, and $\rho(S)(i+1) = \rho(S_0)(i)$, where in each equation, the left hand side is defined just in case the right hand side is.

Similarly, $(+t\,,\ \Phi) \Rightarrow s$ describes S if for some $S_0 \in \Sigma$, s_0 describes S_0 and

1. $\mathsf{tr}(S)(1) = +t$ and $\gamma(S)(1) = \Phi$; and
2. $\mathsf{tr}(S)(i+1) = \mathsf{tr}(S_0)(i)$, $\gamma(S)(i+1) = \gamma(S_0)(i)$, and $\rho(S)(i+1) = \rho(S_0)(i)$, where in each equation, the left hand side is defined just in case the right hand side is.

A strand space for a set of procedures $p_1, \ldots p_n$ is a Σ containing strands described by all the s, υ for which, for some σ, Γ, c, we have $\sigma; \Gamma \vdash \; p_i : s, \upsilon$.

Since σ, σ' are finite partial functions mapping identifiers to values, we write $\sigma \oplus \sigma'$ to mean their disjoint union. That is, if $\sigma \oplus \sigma'$ is defined, then σ, σ' have disjoint domains, and $\sigma \oplus \sigma'$ maps x to a if either σ maps x to a or σ' does.

We use two auxiliary judgments. First, we use the judgment $\Gamma \longrightarrow \phi$ to mean that the formula ϕ is a logical consequence of the formulas Γ. We do not provide inference rules for $\Gamma \longrightarrow \phi$ here; they are inherited from the underlying logic, e.g. Datalog in our implementation. Second, we use the judgment $\Gamma \Vdash \Phi$ to record the successive derivation of the formulas in the list Φ. The values instantiating identifiers appearing free in $\Phi = [\phi_1, \ldots, \phi_n]$ may be chosen left-to-right, in the sense that an implementation may commit to some binding $x \mapsto a$ when x appears free in ϕ_1, even though some later formula ϕ_j may be jointly satisfiable with ϕ_1 only if some other binding $x \mapsto b$ had been chosen. That is, an implementation may get stuck and cause a strand to fail, even when a more farsighted choice of bindings would have made success possible.

Sequential derivation

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\phi_1) \qquad \Gamma \longrightarrow \phi_1\,\sigma \qquad \Gamma \Vdash \Phi\,\sigma_1}{\Gamma \Vdash [\phi_1; \Phi]\,\sigma_1}$$

Vacuous derivation

$$\Gamma \Vdash [\,]$$

Fig. 5. Sequential derivation and instantiation

Structured Operational Semantics. The semantics of procedure p is given by describing its behavior when it is invoked. In this semantics, a procedure is invoked when it receives a message with a call tag, its own principal identity, an activation identifier a, and a vector of atoms, one for each parameter declared by the procedure (Invocation in Figure 6). The initial environment σ_{orig} maps the principal identifier pr to the executing principal's identity.

Invocation

$$\frac{\sigma_1 = \sigma_{orig} \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(pr,\ n,\ ai,\ x^*) \qquad \sigma_1; \Gamma_0, (\Psi\,\sigma) \vdash\ \ c : s, \upsilon}{\sigma_{orig}; \Gamma_0 \vdash\ \ \texttt{proc}\ n\ \Psi\ x^*\ c : (-\mathsf{call}\ pr,\ n,\ ai,\ x^*\,\sigma_1\,,\ \Psi\,\sigma_1) \Rightarrow s, \upsilon}$$

Fig. 6. Procedure Semantics

The procedure semantics show the principal and activation identifier being bound to pr, ai, but this binding is hidden from programmers. In CPPL programs, there is nothing special about the identifiers `pr` and `ai`.

A run of a procedure may conclude by signaling a failure. It does so by sending a message with a fail tag, its principal identifier, and the activation identifier ai. The code c causing a failure may be an empty list of send branches `()`, or a return statement return $\Phi\ x^*$ whose formulas Φ cannot be guaranteed. It may also be a channel name together with an empty list of receive branches `(`x`)`, or else a name that is not bound to a channel, followed by zero or more receive branches,

or else any receive statement that the implementation considers to have timed out. A successful run of a procedure concludes by returning its results, or by invoking a subprotocol by means of a tail recursive call. In this semantics, to return, the strand sends a message with a ret tag, the activation identifier ai, and an atom for each variable named in the return statement (Return, Tail call in Figure 7). The activation identifier ai is used to ensure results are delivered

FAIL

$$\sigma; \Gamma \vdash \ c : \langle (+\mathsf{fail}\, ai\, \sigma\, ,\ \mathsf{true})\rangle, \emptyset$$

RETURN

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi) \qquad \Gamma \Vdash \Phi\, \sigma_1}{\sigma; \Gamma \vdash \ \texttt{return}\ \Phi\ x^* : \langle (+\mathsf{ret}\,(ai,\ x^*)\, \sigma_1\, ,\ \Phi\, \sigma_1)\rangle, \emptyset}$$

TAIL CALL

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi) \qquad \Gamma \Vdash \Phi\, \sigma_1}{\sigma; \Gamma \vdash \ \texttt{(call}\ \Phi\ n\ x^*\ y^*\ \Psi\ \texttt{return}\ \Psi\ y^*\ cb^*\texttt{)} : \langle (+(\mathsf{call}\, pr,\ n,\ ai,\ x^*)\, \sigma_1\, ,\ \Phi\, \sigma_1)\rangle, \emptyset}$$

Fig. 7. Success and Failure Semantics

to the proper caller. To do so, the caller uses a uniquely originating atom, noted in the semantics by adding it to the set of atoms associated with the calling strand (see Figure 11). The "let new" statement generates a nonce or session key, also a uniquely originating atom (Let new in Figure 8). The "let channel" and "let accept" statements also bind the variable x to a value, in this case a channel created by the runtime system. We omit formalizing them. In all of the "let" statements, we require the let-bound identifier not to have been bound previously. In this way we preserve the principle that the environment may be extended with new bindings, but the value bound to any identifier never changes.

LET NEW

$$\frac{a \notin \upsilon \qquad \sigma_1 = \sigma \oplus (x \mapsto a) \qquad \sigma_1; \Gamma \vdash \ c : s, \upsilon}{\sigma; \Gamma \vdash \ \texttt{let}\ x = \texttt{new in}\ c : s, \upsilon \cup \{a\}}$$

Fig. 8. Let new semantics

The semantics of sending and receiving have much in common. A send branch adds an event—consisting of the sent message paired with the guarantee guarding the send—to the front of any behavior of the following statement (Figure 9). 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 a failure if all branches are refused (Figure 7). For a group of rules, σ' assigns values to identifiers occurring free in Φ, i.e. $\mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi)$. This group contains the

Send with guarantee

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi) \qquad \Gamma \Vdash \Phi\,\sigma_1 \qquad \sigma_1; \Gamma \vdash \;\; c : s, \upsilon}{\sigma; \Gamma \vdash \;\; \texttt{(send}\; \Phi\; x\; m\; c\; sb^*\texttt{)} : (+\mathsf{msg}\,(x,\; m)\,\sigma_1\,,\; \Phi\,\sigma_1) \Rightarrow s, \upsilon}$$

Send alternative

$$\frac{\sigma; \Gamma \vdash \;\; \texttt{(}sb^*\texttt{)} : s, \upsilon}{\sigma; \Gamma \vdash \;\; \texttt{(send}\; \Phi\; x\; m\; c\; sb^*\texttt{)} : s, \upsilon}$$

Fig. 9. Semantics of send

successful Send rule with its guarantee, as well as the Return rule and the Tail call rule in Figure 7.

The semantics of a receive statement has the opposite sign (Figure 10). Moreover, in a group of rules, σ' assigns values to identifiers occurring free in the message pattern m, i.e. $\mathsf{dom}(\sigma') \subseteq \mathsf{ide}(x,\; m)$. This group includes the successful Receive rule, as well as procedure invocation in Figure 6, where the pattern m is $pr,\; n,\; ai,\; x^*$.

Receive and rely

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(m) \qquad \sigma_1; \Gamma, \Psi\,\sigma_1 \vdash \;\; c : s, \upsilon}{\sigma; \Gamma \vdash \;\; \texttt{(}x\; \texttt{recv}\; m\; \Psi\; c\; rb^*\texttt{)} : (-\mathsf{msg}\,(x,\; m)\,\sigma_1\,,\; \Psi\,\sigma_1) \Rightarrow s, \upsilon}$$

Receive alternative

$$\frac{\sigma; \Gamma \vdash \;\; \texttt{(}x\; rb^*\texttt{)} : s, \upsilon}{\sigma; \Gamma \vdash \;\; \texttt{(}x\; \texttt{recv}\; m\; \Psi\; c\; rb^*\texttt{)} : s, \upsilon}$$

Fig. 10. Semantics of receive

The semantics of subprotocol call is a combination of a transmission to the callee and a message reception from it (Figure 11). A call may start a subprotocol that eventually fails, in which case execution has not committed to this branch; execution may continue with the next call branch and the unextended environment σ.

In the "Call and rely" production, σ' assigns values to identifiers occurring in Φ, i.e. $\mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi)$, while σ'' assigns values to identifiers occurring free in the pattern $\mathsf{ret}\, ai,\; y^*$, i.e. $\mathsf{dom}(\sigma'') \subseteq \mathsf{ide}(ai,\; y^*) = y^*$. Our implementation assumes that all of the identifiers y^* will be unbound in σ_1, and issues an error message otherwise, but an implementation could allow some of these identifiers already to be bound, in which case the values received in these positions would have to match the values already bound to the identifiers in σ_1.

Definition 2. If $\delta = \mathrm{proc}\; n\; \Phi\; x^*\; c$ is a CPPL procedure declaration, then $[\![\delta]\!]\sigma\Gamma$ is the set of s, υ such that

$$\sigma; \Gamma \vdash \;\; \delta : s, \upsilon$$

is derivable using the inference rules in this section and the rules for the underlying logic's consequence relation $\longrightarrow$.

Call and rely

$$\frac{\Gamma \Vdash \Phi\,\sigma_1 \qquad \begin{array}{c}\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi) \\ \sigma_2 = \sigma_1 \oplus \sigma'' \qquad \mathsf{dom}(\sigma'') \subseteq \mathsf{ide}(y^*)\end{array} \qquad \sigma_2; \Gamma, \Psi\,\sigma_2 \vdash\ c : s, \upsilon}{\begin{array}{l}\sigma; \Gamma \vdash\ \texttt{(call}\ \Phi\ n\ x^*\ y^*\ \Psi\ c\ cb^*\texttt{)}\ : \\ \quad (+\mathsf{call}\,pr,\ n,\ ai,\ x^*\,\sigma_1\ ,\ \Phi\,\sigma_1) \Rightarrow (-(\mathsf{ret}\,ai,\ y^*\,\sigma'')\ ,\ \Psi\,\sigma_2) \Rightarrow s, \upsilon \cup \{ai\}\end{array}}$$

Callee fails

$$\frac{\sigma_1 = \sigma \oplus \sigma' \qquad \mathsf{dom}(\sigma') \subseteq \mathsf{ide}(\Phi) \qquad \Gamma \Vdash \Phi\,\sigma_1 \qquad \sigma; \Gamma \vdash\ \texttt{(}cb^*\texttt{)}\ : s, \upsilon}{\begin{array}{l}\sigma; \Gamma \vdash\ \texttt{(call}\ \Phi\ n\ x^*\ y^*\ \Psi\ c\ cb^*\texttt{)}\ : \\ \quad (+(\mathsf{call}\,pr,\ n,\ ai,\ x^*\,\sigma_1)\ ,\ \Phi\,\sigma_1) \Rightarrow (-\mathsf{fail}\,ai\ ,\ \mathsf{true}) \Rightarrow s, \upsilon \cup \{ai\}\end{array}}$$

Call alternative

$$\frac{\sigma; \Gamma \vdash\ \texttt{(}cb^*\texttt{)}\ : s, \upsilon}{\sigma; \Gamma \vdash\ \texttt{(call}\ \Phi\ p\ x^*\ y^*\ \Psi\ c\ cb^*\texttt{)}\ : s, \upsilon}$$

Fig. 11. Call Semantics

Given CPPL procedures $\delta_1, \ldots, \delta_n$, let

$$\Delta = \bigcup_{\delta_i, \sigma, \Gamma} [\![\delta_i]\!]\sigma\Gamma.$$

Σ, an annotated strand space with uniqueness assumptions, *models* the procedures $\delta_1, \ldots, \delta_n$ if every $S \in \Sigma$ is described by some s with $s, \Upsilon(S) \in \Delta$, and for every $s, \upsilon \in \Delta$, s describes at least one $S \in \Sigma$ with $\Upsilon(S) = \upsilon$.

Parametric Strands. The structured operational semantics that we have just given clarifies the relations between the code being executed, the runtime environment, the theory in force, and the actions taken. However, there is another kind of regularity in the behavior of CPPL programs. This is the fact that the infinite number of strands described by the semantics are in fact all instances of a finite number of genuinely different strands. They are simply instantiated with infinitely many different values.

Any execution of the return statement $\texttt{return}\ \Phi\ x^*$ unleashes either a strand of the form

$$\langle(+\mathsf{ret}\,(ai,\ x^*)\,\sigma\ ,\ \Phi\,\sigma)\rangle, \emptyset$$

or one of the form $\langle(+\mathsf{fail}\,ai\,\sigma\ ,\ \mathsf{true})\rangle, \emptyset$. If we let σ_0 be an assignment that maps each identifier in this code statement to a value of the appropriate type, then every assignment σ in these two forms may be written as $\sigma_0 \circ \alpha$ for some replacement α. That is, every strand of the forms shown is an instance of the strands for the specific value $\sigma = \sigma_0$. Similarly, any strand unleashed from $\texttt{let}$ $x = \texttt{new in}\ c$ will be of the form $s, \upsilon \cup \{a\}$ for some $a \notin \upsilon$ where s, υ is a strand unleashed from c.

Send and receive branches are roughly tagged unions. The strands that may be unleashed by the send branches $\texttt{(}sb^*\texttt{)}$ are, in addition to a failure, all strands that may be unleashed by the code embedded within the send branches, each prefixed with a single positive message pattern. For receive branches, the prefixed

pattern is negative. However, our nondeterministic semantics does not require the "tagging" initial patterns to be disjoint. Call branches are slightly more complex, since there is the uncommitted behavior of a call and a failure, preceding invocation of another branch.

We refer to this informally presented finite set of strands as $\mathcal{S}$. Suppose in a procedure δ the *nesting depth* d is the number of nested parentheses introduced by send, receive, and call statements. Let the *branching factor* k be the maximum number of send branches, receive branches, and call branches in any one statement.

Proposition 1. *Suppose the depth of a procedure δ is d and its branching factor is k. There is a set $\mathcal{S}(\delta)$ of strands with cardinality $|\mathcal{S}(\delta)| \leq k^d$ such that, for every strand s, υ, if $s, \upsilon \in [\![\delta]\!]\sigma\Gamma$, then $s, \upsilon = (s_0, \upsilon_0) \cdot \alpha$ for some α and some $s_0, \upsilon_0 \in \mathcal{S}(\delta)$.*

If $s, \upsilon = (s_0, \upsilon_0) \cdot \alpha$ for some α and some $s_0, \upsilon_0 \in \mathcal{S}(\delta)$, then for some σ, Γ, $s, \upsilon \in [\![\delta]\!]\sigma\Gamma$. If $s_0, \upsilon_0 \in \mathcal{S}(\delta)$, then $\mathsf{length}(s_0) \leq (2kd) + 2$.

Typically, k and d are small, and the cardinality of $\mathcal{S}(\delta)$ is far less than k^d. Although a finite set of procedures δ_i yields a finite set $\bigcup \mathcal{S}(\delta_i)$, there are nevertheless infinitely many global executions associated with the δ_i; indeed, natural questions such as secrecy are not uniformly decidable [16], although important classes are decidable [6, 22].

We assume that for every replacement α, $\Gamma \longrightarrow \phi$ implies $\Gamma \cdot \alpha \longrightarrow \phi \cdot \alpha$, this being a defining property of a consequence relation for logics with replacements.

Proposition 2. *For every procedure δ, and replacement α, the judgment*

$$\sigma; \Gamma \vdash \ \delta : s, \upsilon \quad \textit{implies} \quad \sigma \cdot \alpha; \Gamma \cdot \alpha \vdash \ \delta : (s, \upsilon) \cdot \alpha.$$

Proof. Each rule is invariant under applying a replacement α.

5 Global Semantics

In order to model subprotocol call and return, and other local or inherently secure interactions, we enrich the notion of a direction. Directions will distinguish transmission from reception as before. However, a direction may additionally specify that the peer at the other end of a message transmission arrow is regular. It may also specify, in the case of message transmission, that the message will definitely be delivered.

Definition 3. A *direction* d is a value with the following properties: (1) the *polarity* of d is one of the symbols $+, -$, indicating transmission and reception respectively; (2) the *partner* of d is one of the symbols *regular* and *any*; and (3) the delivery *confidence* of d is one of the symbols *guaranteed* and *maybe*.

We write directions in the form $+_p^c$ and $-_p^c$. The subscript p indicates whether the partner is regular (r) or any (a). The superscript c indicates whether the

delivery confidence is guaranteed (g) or maybe (m). When the partner is any, we generally omit the subscript. When the delivery confidence is maybe, we generally omit the superscript. We say that a node is *negative* when its polarity is $-$, and that it is positive when its polarity is $+$. The delivery confidence is of interest only when a node is positive; the recipient of a message knows that it has been received. With this amplification of the notion of direction, we preserve the definitions of strand space from the beginning of Section 4.

Strands are either *penetrator strands*, taking the forms shown in Definition 9 from Appendix A, or else substitution instances of a finite number of *roles* of a given protocol. When a protocol is defined by a finite number of CPPL declarations $\delta_1, \ldots, \delta_n$, then these roles are the members of $\bigcup \mathcal{S}(\delta_i)$ as in Proposition 1. We call the instances of the roles *regular strands*.

Transmission that preserves confidentiality is a special kind of message transmission; these nodes have direction d with positive polarity and regular partner. Reception that ensures authenticity is (dually) a special kind of message reception; these nodes have direction d with negative polarity and regular partner. If a communication arrow $n \to n'$ ensures both confidentiality and authentication, then the directions of n and n' both have regular partner. Purely local communication such as subprotocol call or return is of this kind.

We write Conf for the set of nodes n of the form $+_r^c$ and Auth for the set of nodes n of the form $-_r^c$ (where c may be either g or m).

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 8). The content of the annotations comes from an enriched notion of bundle, in which message transmission arrows $n_1 \to n_2$ behave as indicated by the properties of the directions of the two nodes.

Definition 4. Let $\mathcal{B} = \langle \mathcal{N}_\mathcal{B}, (\to_\mathcal{B} \cup \Rightarrow_\mathcal{B}) \rangle$ be a finite acyclic subgraph of $\langle \mathcal{N}, (\to \cup \Rightarrow) \rangle$. $\mathcal{B}$ is a *bundle with secure communication* or *sc-bundle* if:

1. If $n_2 \in \mathcal{N}_\mathcal{B}$ is negative, then there is a unique n_1 such that $n_1 \to_\mathcal{B} n_2$.
2. If $n_2 \in \mathcal{N}_\mathcal{B}$ and $n_1 \Rightarrow n_2$ then $n_1 \Rightarrow_\mathcal{B} n_2$.
3. If $n' \in \mathsf{Auth}$ and $n \to_\mathcal{B} n'$, then n is regular. If $n \in \mathsf{Conf}$ and $n \to_\mathcal{B} n'$, then n' is regular; if moreover $n \to_\mathcal{B} n''$, then $n' = n''$.
4. If $n_1 \in \mathcal{N}_\mathcal{B}$ is positive with delivery confidence *guaranteed*, then there is a n_2 such that $n_1 \to_\mathcal{B} n_2$.

$n \preceq_\mathcal{B} n'$ if some sequence of zero or more arrows $\to_\mathcal{B}, \Rightarrow_\mathcal{B}$ lead from n to n'.

An sc-bundle $\mathcal{B}$ does not assume secure communication if every node n occurring in it has partner *any* and delivery confidence *maybe*. Thus, the bundles in the sense of earlier work such as [21] are a special case of sc-bundles.

Proposition 3. *If $\mathcal{B}$ is an sc-bundle, $\preceq_\mathcal{B}$ is a finite partial order. Every non-empty subset of the nodes in $\mathcal{B}$ has $\preceq_\mathcal{B}$-minimal members.*

Secure Communication within CPPL. We represent the strands in the CPPL semantics as strands with secure communication as a function of the tags in the terms. In particular, if a term is of one of the forms $\pm\mathsf{call}\, t$, $\pm\mathsf{ret}\, t$, or $\pm\mathsf{fail}\, t$, then we regard the direction as being $\pm^g_r$. That is, the partner is assumed regular, and the delivery is assumed to be guaranteed. If a term is of the form $\pm\mathsf{msg}\, t$, then we regard the direction as being $\pm^m_a$. As a consequence, any sc-bundle formed from CPPL strands will provide authentication, confidentiality, and guarantee of delivery for the local mechanism of subprotocol invocation and termination. No assumption is made for the messages dispatched and received in ordinary protocol transmission and reception events.

In this, we follow the Dolev-Yao model for protocol messages over the network, but we assume that each individual participant has a secure platform on which to run her CPPL procedures. Secure communication in the sense of Definition 4 can also be used to represent communication through a secure transport medium such as the tunnels provided by TLS and IPsec, thus providing a strand space variant to the methods of Broadfoot and Lowe [9]. We will now develop a method—encapsulated in Propositions 4–6 and the finite semantics $\mathcal{S}(\delta)$—to prove security properties for the procedure definitions of a protocol.

6 Reasoning about the Global Semantics

Occurrences and Sets. We view each term as an abstract syntax tree, in which atoms are leaves and internal nodes are either *tagged messages*, *concatenations*, or else *encryptions*. A branch through the tree *traverses a key child* if the branch traverses an encryption $\{|t|\}_K$ and then traverses the second child (the key) labeled K.

An *occurrence* of t_0 in t is a branch within the tree for t that ends at a node labeled t_0 without traversing a key child. A *use* of K in t (for encryption) is a branch within the tree for t that ends at a node labeled K and that has traversed a key child. We say that t_0 is a *subterm* of t (written $t_0 \sqsubset t$; see Definition 8, Clause 1) if there is an occurrence of t_0 within t. When S is a set of terms, t_0 *occurs only within* S in t if, in the abstract syntax tree of t, every occurrence of t_0 traverses a node labeled with some $t_1 \in S$ (properly) before reaching t_0. Term t_0 *occurs outside* S in t if $t_0 \sqsubset t$ but t_0 does not occur only within S in t.

A term t *originates* at node n_1 if n_1 is positive, $t \sqsubset \mathrm{term}(n_1)$, and $n_0 \Rightarrow^+ n_1$ implies $t \not\sqsubset \mathrm{term}(n_0)$. It *originates uniquely* in a set N of nodes if there is exactly one $n \in N$ at which it originates. It is *non-originating* in N if there is no $n \in N$ at which it originates.

Definition 5 (Safety). Let $\mathcal{B}$ be an sc-bundle. $a \in \mathsf{Safe_ind}_0(\mathcal{B})$ if a originates nowhere in $\mathcal{B}$. $a \in \mathsf{Safe_ind}_{i+1}(\mathcal{B})$ if either (1) $a \in \mathsf{Safe_ind}_i(\mathcal{B})$ or else

(2) a originates uniquely on a regular node $n_0 \in \mathcal{B}$ and, for every positive regular node $n \in \mathcal{B}$ such that $a \sqsubset \mathrm{term}(n)$, the following holds: Either $n \in \mathsf{Conf}$ or else a occurs only within S in $\mathrm{term}(n)$, where

$$S = \{ \ \{|h|\}_{K_0} \colon K_0^{-1} \in \mathsf{Safe_ind}_i(\mathcal{B}) \ \}.$$

$a \in \mathsf{Safe_ind}(\mathcal{B})$ if there exists an i such that $a \in \mathsf{Safe_ind}_i(\mathcal{B})$.

Proposition 4 (Safety ensures secrecy). *If $a \in \mathsf{Safe_ind}(\mathcal{B})$ and there exists $n \in \mathcal{B}$ such that term$(n) = a$, then n is regular.*

Proofs of this proposition and the others in this section will appear elsewhere.

Definition 6 (Export Protection). A set S of terms provides *export protection* for $\mathcal{B}$ if for every $t \in S$, t is of the form $\{|h|\}_K$ where $K^{-1} \in \mathsf{Safe_ind}(\mathcal{B})$.

When C is a set of terms, we also write $\mathsf{Conf}(C)$ for the set of nodes $n \in \mathsf{Conf}$ such that term$(n) \in C$. The outgoing authentication test allows us to infer that there is a *regular* strand including $m_0 \Rightarrow^+ m_1$ as in Figure 12.

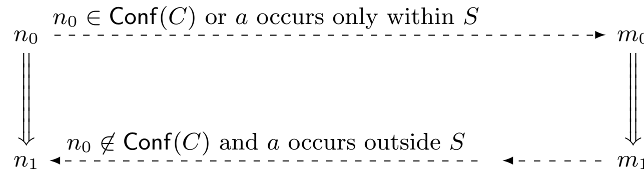

Fig. 12. The Outgoing Authentication Test

Proposition 5 (Outgoing Authentication Test). *Let $\mathcal{B}$ be an sc-bundle with regular nodes $n_0, n_1 \in \mathcal{B}$; let S be a set of terms providing export protection for $\mathcal{B}$; and let C be a set of terms. Suppose that (1) a originates uniquely at n_0 and either $n_0 \in \mathsf{Conf}(C)$ or else a occurs only within S in term(n_0); and (2) $n_1 \notin \mathsf{Conf}(C)$ and a occurs outside S in term(n_1).*

There exists a regular $m_0 \Rightarrow^+ m_1$ such that (1) m_0 is the earliest occurrence of a on its strand s; (2) m_1 is the earliest node on s such that $m_1 \notin \mathsf{Conf}(C)$ and a occurs outside S in term(m_1); (3) m_1 is positive, and m_0 is negative unless $m_0 = n_0$. Moreover, $n_0 \preceq_{\mathcal{B}} m_0 \Rightarrow^+ m_1 \preceq_{\mathcal{B}} n_1$; $a \sqsubset$ term(m_0); and for all $m \preceq_{\mathcal{B}} m_0$, either $n_0 \in \mathsf{Conf}(C)$ or a occurs only within S in m.

Proposition 6 (Incoming Authentication Test). *Suppose $n_1 \in \mathcal{B}$ is negative. (1) If $t \sqsubset$ term(n_1) and $t = \{|h|\}_K$ for $K \in \mathsf{Safe_ind}(\mathcal{B})$, then there exists a positive regular $m_1 \prec n_1$ such that t originates at m_1. (2) If $n_1 \in \mathsf{Auth}$, then there exists a unique positive regular $m_1 \rightarrow n_1$. Moreover in either case:*

Solicited Incoming Test *If $a \sqsubset t$ originates uniquely on $n_0 \neq m_1$, then $n_0 \preceq m_0 \Rightarrow^+ m_1 \prec n_1$.*

Propositions 4–6 suffice to prove the main authentication and secrecy properties of protocols. In our context, we want particularly to establish *soundness*, i.e. that in every execution, one principal's relies are supported by earlier guarantees by other principals [23]. We write $\mathsf{prin}(m)$ to refer to the regular principal acting on a node m, which we assume is some conventionally chosen parameter to the regular strand that m lies on. If m lies on a penetrator strand, then $\mathsf{prin}(m)$ is undefined. We also write P says ϕ, subject to the understanding that this will be encoded suitably into the implemented logic.

Definition 7. Soundness. Bundle $\mathcal{B}$ *supports* a negative node $n \in \mathcal{B}$ iff $\rho(n)$ is a logical consequence of the set of formulas $\{\mathsf{prin}(m)\ \mathsf{says}\ \gamma(m) \colon m \prec_{\mathcal{B}} n\}$.

Let Π be an annotated protocol, and let $\mathbb{B}$ be a set of sc-bundles over Π. Π *is sound for* $\mathbb{B}$ if, for all $\mathcal{B} \in \mathbb{B}$, for every negative $n \in \mathcal{B}$, $\mathcal{B}$ supports n.

In practice, we use the authentication test theorems to prove the existence of nodes m such that the formulas $\mathsf{prin}(m)\ \mathsf{says}\ \gamma(m)$ imply $\rho(n)$. Since only positive regular nodes m help to support $\rho(n)$, if we cannot prove the existence of positive regular nodes m of a protocol preceding a negative node n, then the rely formula on n must be trivial, i.e. a consequence of the empty set of formulas. In particular if the message received on n could have been generated without help by the adversary, then $\rho(n) = [\,]$, i.e. it is vacuously true.

7 Example: Protocol-based Access

We will now return to the NS Quote Protocol given at the beginning of the paper in Figure 1. In it, an initiator A requests on-line stock quotes from a responder B, and B delivers them if it can determine that A is a registered subscriber. We prove first that it is unsound, hardly surprising as it is based on the (broken) Needham-Schroeder protocol.

Proposition 7. *NSQ is unsound; there is a bundle $\mathcal{B}$ in which the public keys of A and B are non-originating, and the nonces N_a, N_b are uniquely originating, but in which node n_4 is unsupported.*

Proof. In Figure 13, $\rho(n_4) = A\ \mathsf{says}\ \mathsf{requests}(A, B, D)$, while by contrast $\gamma(m_1) = \mathsf{requests}(A, M, D')$.

A Corrected Protocol. To correct the protocol, and also enrich its functionality, we revise our example protocol to take the form in Figure 14. In this version of the protocol, we add B's name to the message sent from node n_2, as suggested by Lowe [29] when he discovered the attack we showed in Figure 13. We also add a decision, made by the server B. It chooses in nodes n_4 and n'_4 between two levels of service. Corporate users may pay dear, but they receive prompt delivery of precise data at a premium price; individual users may pay much more cheaply to receive information that is delayed a few minutes and rounded from thirty-seconds of a point to the nearest eighth of point. The resulting protocol is

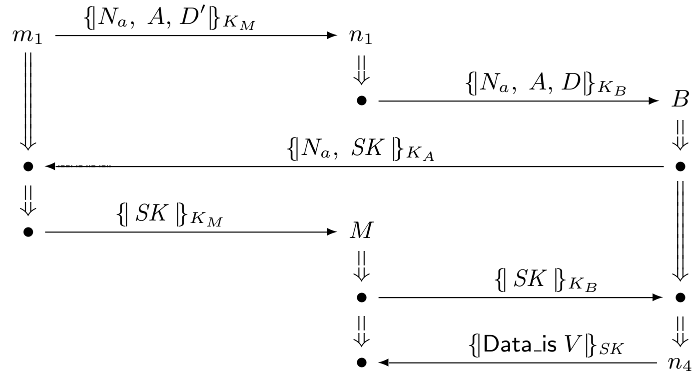

Fig. 13. Counterexample to NS Quote Soundness

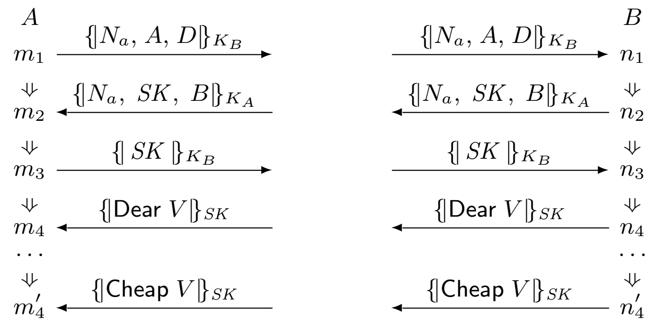

Fig. 14. NSL Quote Protocol with Choice

the same except at the last step, where different tags distinguish the two types of outcome (Figure 14). There are three steps we will take to implement this example. First, we will program the message flow, namely the portion of the CPPL implementation that manipulates communication channels, generates nonces and session keys, and sends and receives messages. Second, we will integrate the trust management semantics for each of the messages. The final step is to specify procedure headers, thereby providing a way to link behaviors together by calling subprotocols. In this example, the benefit is to allow flexibility in retrieval of certified public keys. However, the general ability to encapsulate subprotocols in an informative way appears to us to be one of the major strengths of the CPPL integration of trust management and protocols.

Message Behavior. The client generates a nonce, opens a channel to the server, and then expects to engage in two round trips of sending a message and receiving a reply (Figure 15). The `return` statements here do not carry the parameters and final guarantee, because those will be declared instead in the procedure header. We omit the trust management formulas for now, leaving only underscores in their place.

```
let chan = connect(b_addr) in
let na = new nonce in
  send _ chan {na, a, d} kb
  receive chan {na, k, b} ka _
  send _ chan {k} kb
  receive chan cases
    {Dear v} k _  return
  | {Cheap v} k _ return
end
```

Fig. 15. Client Behavior in NSL Quote

The server (Figure 16) waits to accept an incoming connection. It then receives a message off that channel, authenticates the claimed sender via a message round trip, and delivers data of one quality or the other, tagged with either Dear or Cheap.

Trust Management Annotations. For readability, we italicize the trust management formulas. The client (Figure 17) guarantees that it is requesting the information in its first outgoing message, and relies on the server having guaranteed the information in its last incoming message, at one of the two possible levels of service.

The server's trust management behavior is described in Figure 18. The rely formula on the server's first receive statement is empty, i.e. an empty list `[]` meaning *true*, as is required because the adversary may have prepared the

```
let chan = accept in
  receive chan {na, a, d} kb _
  let k = new symkey in
    send _ chan {na, k, b} ka
    receive chan {k} kb _
    send cases
      _ chan {Dear v} k  return
    | _ chan {Cheap v} k return
end
```

Fig. 16. Server Behavior in NSL Quote

```
let chan = connect(b_addr) in
let na = new nonce in
  send [requests(a,b,d)] chan {na, a, d} kb
  receive chan {na, k, b} ka []
  send [] chan {k} kb
  receive chan cases
    {Dear v} k [says_curr_val(b, d, v)]    return
  | {Cheap v} k [says_approx_val(b, d, v)] return
end
```

Fig. 17. Client Trust Management, NSL Quote

message `{na, a, d} kb`. Before the final transmission, the server chooses between the two branches in the send statement according to a trust management formula, guaranteeing payment for the information transmitted, and retrieving a current value for `v`. If B can establish that A will pay for the high quality data it selects the first branch. If B can establish only that A will pay for the low quality data, it selects the second branch. If B cannot establish even that, for instance because A is not yet a subscriber, then B must fail in this protocol run, terminating abnormally without sending either of these messages, and without returning information to its caller. For either class of service, part of determining whether A will pay for the data is determining whether he has requested it. A crucial authentication service provided by the protocol is to justify B in relying on this conclusion when B receives message three, illustrating the value of protocol soundness. B has one other guarantee in this version of the protocol; he guarantees `owns(a,ka)` asserting that the purported peer in this run is the owner of the public encryption key to be used in this run. This is the first occurrence of the identifier `ka`, reflecting the fact that the guarantee is a query, in the manner of logic programming; it binds the new identifier `ka` to some value k for which the trust management engine can establish that the principal bound to `a` owns k as public encryption key.

Procedure Headers. We encapsulate the behavior of CPPL procedures using headers. The header gives the name of the procedure, the parameters with which it

```
let chan = accept in
  receive chan {na, a, d} kb []
  let k = new symkey in
    send [owns(a, ka)] chan {na, k, b} ka
    receive chan {k} kb [says_requests(a, a, b, d)]
    send cases
      [will_pay_dear(a, d); curr_val(d, v)] chan {Dear v} k
      return
    | [will_pay_cheap(a, d); approx_val(d, v)] chan {Cheap v} k
      return
end
```

Fig. 18. Server Trust Management, NSL Quote

should be called, the parameters that it will return, and two formulas. The first, the *rely* statement, declares the condition under which this procedure may properly be called. It is a relation on the parameters to the call. The caller must guarantee at least this strong a condition before calling the procedure with actual parameters. The second statement is the procedure's *guarantee*. This is a relation on the procedure's input and output parameters, and it defines what the caller has learned by means of the procedure call. The guarantee need only hold on successful termination; failure returns no parameters and guarantees no formula. The server's guarantee *supplied(a, q, d, v)* informs its caller

```
client (a, ka, b, kb, b_addr, d) (na, q, v)
    rely [owns(a, ka); owns(b, kb); at(b, b_addr)]
    guarantee [val(d, q, v)]
  statement, see Figure 17
end

server (b:text, kb) (a, q, d, v)
    rely [owns(b, kb)]
    guarantee [supplied(a, q, d, v)]
  statement, see Figure 18
end
```

Fig. 19. Client and Server Procedure Headers in NSL Quote

that data has been supplied to a client, so that the client may be billed. The identifier `q` is one of the return parameters; the participants use it to interpret the quality of the information returned in `v`. The identifier `q` occurs only in the return guarantee, and the trust management engine selects a suitable value for it immediately before a successful return. It uses the rules in Figure 20 as an ingredient in selecting its value. The client must additionally use its trust in `b`

```
val(D, "high quality", V) :-
  curr_val(D, V).

val(D, "low quality", V) :-
  approx_val(D, V).
```

Fig. 20. Axioms Governing Quality for NSL Quote

for this type of data, inferring `curr_val(d, V)` from `says_curr_val(b, d, V)`, and inferring `approx_val(d, V)` from `says_approx_val(b, d, V)`.

Subprotocols. An advantage of connecting procedures with their trust management pre- and post-conditions in this way is that it leads to attractive notions of subprotocol and of call. We illustrate subprotocols here (Figure 21) by incorporating an optional subprotocol for certificate retrieval when the server B does not have a certificate for the client in its local certificate database. Possibly B would like to increase its clientele; an independent service certifies customers, and delivers the certificates for a fee. B attempts to retrieve the client's public key from its local store; if that succeeds, it calls the `null_protocol`, which does nothing. If the local retrieval fails, it consults the certification service via the `get_public_key` protocol. This protocol may be implemented separately, as the only constraint that the programmer requires is that it should satisfy the interface given in its header. We summarize the protocol's correctness in a soundness

```
maybe_get_public_key (a:text) (ka:verkey)
    guarantee [owns(a, ka)]
  call cases
    [owns(a, ka)]
      null_protocol () () []
    return
  | [cert_auth(c:text); owns(c, kc:verkey); at(c, c_addr:text)]
      get_public_key (a, c, kc, c_addr) (ka)
      [owns(a, ka)]
    return
end

null_protocol () () return end

get_public_key (a:text, c:text, kc, c_addr) (ka:verkey)
    rely [owns(c, kc); at(c, c_addr)]
    guarantee [owns(a, ka)]
  ... statement ...
end
```

Fig. 21. Subprotocols for Certificate Retrieval

assertion. We state it here without being precise about the unique origination and non-origination assumptions that define the set $\mathbb{B}$ of bundles with respect to which soundness holds.

Proposition 8. *The set of* CPPL *procedures displayed in Figures 17–21 is sound for bundles in which the private decryption keys are uncompromised and the newly generated values are uniquely originating.*

Using the disjoint encryption result for protocol composition [20], the soundness of the main protocol depends only on a simple property of the certificate retrieval subprotocol, beyond what is declared in its header.

8 The Current CPPL Implementation

We have developed two successive CPPL implementations. The second generation compiler was written after the structured operational semantics presented here in Section 4, and benefited from the concise specification. In both cases, we used OCaml [27] as the implementation language, and the compilers translate a CPPL source file into OCaml. When parsing a source file, the compiler generates an abstract syntax tree modeled after the core syntax given in Figure 2. In particular, it replicates the return parameter list and the guarantee formula in the header for a procedure at each `return` statement within that procedure. Each CPPL procedure is translated into an OCaml procedure that takes a number of CPPL values as arguments and returns a tuple of results.

A full CPPL program is constructed from at least two source files. The first is a CPPL source file used to specify the CPPL procedures. The other is an OCaml source file that defines the main routine invoked when the program is started. This routine generates the principal's theory from a sequence of Datalog [10] formulas, and generates additional Datalog facts by opening a keystore containing public keys. Code generated from both files is linked against three libraries needed at runtime. One is a communications library. It provides the channel abstraction, including code to open channels to specified addresses and to await an incoming connection. Second, the cryptographic library controls the formatting of messages as bitstrings, and provides abstractions of keys and operations for encryption, decryption, hashing, signatures, and verification. Because of the well-defined interfaces, alternative libraries can easily be substituted; we have developed one cryptographic library based on Leroy's Cryptokit [26] and another that provides access to a Trusted Platform Module [35, 36], if the latter is available on the underlying hardware.

The third main runtime library is our Datalog [10] trust management engine. Datalog is a declarative logic language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause. Our implementation uses the tabled logic programming algorithm described in [11, 12]. All queries terminate because of Datalog's syntactic restrictions, and because the implementation maintains a table of intermediate results.

One of the main jobs of the compiler is to translate the message patterns contained in the CPPL source program into executable code. For a message pattern in a receive statement, the generated code must parse incoming messages. The compiler emits code containing calls to the interface procedures exported by the cryptographic library. The emitted code must raise an exception if the incoming message is not of the right form, or if an identifier in the pattern is already bound, and the incoming message contains a different value in that position from the value bound in the runtime environment. For message patterns in transmission statements, the generated code must use the cryptographic library to assemble a suitable concrete message, which is then handed to the communications library for transmission through a channel. When asymmetric cryptography is used in message transmission or reception, the code may require the cryptographic library to use a private key held only in its own keystore. For instance, when the NSL Quote server B parses the message `{na, a, d}kb`, `kb` is bound in the runtime environment. However, parsing succeeds only if the cryptographic library possesses a private decryption key K_B^{-1} inverse to the value K_B bound to `kb`. The private decryption key is not mentioned in CPPL source programs, and it is the obligation of the cryptographic library together with the main routine to ensure that suitable private keys are available. The semantic productions for Send and Receive in Figures 9 and 10 are optimistic, since they do not indicate that these keys may be missing. The Receive production also does not explicitly say that the environment is extended only from values contained as subterms of an incoming message, not from values used only as encryption keys.

A number of different demonstration-sized protocols have been implemented in CPPL, suggesting solutions to different information security problems. Alternative protocols allow adapting the solutions to differing trust relations among the principals.

9 Conclusion

Three central ideas have shaped our approach to CPPL. First, cryptographic protocols are a coordination mechanism between principals. The purpose of a cryptographic protocol is to ensure that principals which have successfully completed their strands are sure to agree on certain values [37, 30]. In this view, an authentication property is an assertion about parameters matching between separate strands. “Entity authentication” means agreeing specifically on the parameters naming the principals; “message authentication” for a message t requires agreement on all of the atomic values contained in t. These and other variants of agreement may be proved uniformly using the authentication test theorems (Propositions 4–6). In the case of an annotated protocol, in which nodes are associated with rely and guarantee formulas, an agreement on values also ensures a corresponding degree of agreement between the principals on assertions; we summarize this in the notion of soundness (Definition 7). Many protocols must also establish recency, by ensuring that an event in each local run occurs between two events of each other local run [19]. Recency comes for free from the outgoing

test and the solicited incoming test. A cryptographic protocol thus coordinates values, assertions, and time across different strands.

Our second motivating idea was that trust decisions at run time may control a principal's protocol behavior. Each message transmission is associated with a *commitment* that the principal makes if it transmits the message. If the principal cannot derive the trust constraint for a message, then the principal does not send the message. This provides a mechanism for selecting between branches of execution, namely to choose a branch with a derivable guarantee. If there is no such branch, then the principal stops and aborts this protocol run.

Our third central idea was a semantic idea. We gave the semantics for a single protocol procedure as a finite number of parametric strands, each of bounded length. Each regular strand determines a sequence of messages that may have been sent and received by the time the run is complete; these messages are parametrized by the values (keys, nonces, names, prices, etc.) selected in this run. The instances of the parametric strands are determined by the structured operational semantics presented in Section 4. A global execution is a *bundle*. This says that it is a number of regular strands, possibly together with penetrator strands, that are linked together in a causally well founded way (Section 5). A regular (non-penetrator) strand in a bundle represents a sequence of transmissions and receptions enacted by one principal while executing a single session of a single protocol role or subprotocol role. The bundle may use *secure communication*, allowing it to model subprotocol call and return as local, secure message transmissions. In Section 6 we developed useful techniques for determining whether all bundles for a particular set of protocols and subprotocols satisfy security goals.

Future Work. Various areas remain for future work. For instance, our method carefully separates the protocol properties that are used to prove soundness, but not represented in a logic, from the trust management decisions that are logically represented. The advantage of this procedure is that there is a clear boundary between operational reasoning about protocol behavior and logical reasoning within trust theories. However, there is also a disadvantage, since reasoning involving both protocol behavior and its trust consequences is not easily integrated. As an example, if a principal is deciding whether to accept a new protocol, it would be desirable to deduce its acceptability from an explicit policy. We also need a better way to represent the imperative effects that may be the result of a protocol execution, for instance, a bank transferring money from buyer to seller at the end of an electronic commerce transaction. Finally, the current data model of CPPL is extremely impoverished, and an improved language would allow processing of structured data to be integrated with protocol actions and trust decisions.

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. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.
6. Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols: Tagging enforces termination. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, number 2620 in LNCS, pages 136–152. Springer, April 2003.
7. Matt Blaze, Joan Feigenbaum, and Jack Lacy. Distributed trust management. In *Proceedings, 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1997.
8. Michele Boreale. Symbolic trace analysis of cryptographic protocols. In *ICALP*, 2001.
9. 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.
10. Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions of Knowledge and Data Engineering*, 1(1), 1989.
11. W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Logic Prog.*, 24(3):161–199, 1995.
12. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
13. Federico Crazzolara and Giuseppe Milicia. Developing security protocols in χ-spaces. In *Proceedings, 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.
14. 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.
15. Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
16. Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004. Initial version appeared in *Workshop on Formal Methods and Security Protocols*, 1999.
17. Cédric Fournet, Andrew Gordon, and Sergei Maffeis. A type discipline for authorization policies. In Mooly Sagiv, editor, *European Symposium on Programming*, volume LNCS No of *LNCS*. Springer Verlag, 2005.
18. 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.

19. Joshua D. Guttman. Key compromise and the authentication tests. *Electronic Notes in Theoretical Computer Science*, 47, 2001. Editor, M. Mislove. URL `http://www.elsevier.nl/locate/entcs/volume47.html`, 21 pages.
20. Joshua D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. In *Proceedings, 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, July 2000.
21. Joshua D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.
22. Joshua D. Guttman and F. Javier Thayer. The sizes of skeletons: Decidable cryptographic protocol authentication and secrecy goals. MTR 05B09 Revision 1, The MITRE Corporation, March 2005.
23. 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.
24. Charlie Kaufman, ed. Internet key exchange (IKEv2) protocol. Internet Draft, September 2004. Available at `http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ikev2-17.txt`.
25. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
26. Xavier Leroy. Cryptokit. Sofwtare available via `http://pauillac.inria.fr/~xleroy/software.html`, April 2005. Version 1.3.
27. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System*. INRIA, `http://caml.inria.fr/`, 2000. Version 3.00.
28. 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.
29. Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.
30. Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop Proceedings*, pages 31–43. IEEE Computer Society Press, 1997.
31. Jonathan Millen and Frederic Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.
32. Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.
33. 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.
34. 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.
35. Trusted Computing Group, `https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Ov%erview.pdf`. *TCG Specification Architecture Overview*, revision 1.2 edition, April 2004.
36. Trusted Computing Group, `https://www.trustedcomputinggroup.org/downloads/specifications/mainP1DP%_rev85.zip`. *TPM Main: Part I Design Principles*, specification version 1.2, revision 85 edition, February 2005.

37. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.

A Additional Strand Notions

Definition 8. 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 $t \sqsubset g,\ h$ 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 $1 \leq i \leq \mathsf{length}(\mathrm{tr}(s))$. 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 $n_1 \rightarrow n_2$ iff $\mathrm{term}(n_1) = +t$ or $+_c\ t$ and $\mathrm{term}(n_2) = -t$ 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 $j \geq i$ (respectively, $j > i$).
4. Suppose I is a set of terms. The node $n \in \mathcal{N}$ is an *entry point* for I iff $\mathrm{term}(n) = +t$ for some $t \in I$, and whenever $n' \Rightarrow^+ n$, $\mathrm{term}(n') \notin I$. 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 9. 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 $K \in \mathsf{K}_{\mathcal{P}}$
$\mathsf{C}_{g,h}$: $\langle -g,\ -h,\ +g,\ h\rangle$
$\mathsf{S}_{g,h}$: $\langle -g,\ h,\ +g,\ +h\rangle$
$\mathsf{E}_{h,K}$: $\langle -K,\ -h,\ +\{|h|\}_K\rangle$
$\mathsf{D}_{h,K}$: $\langle -K^{-1},\ -\{|h|\}_K,\ +h\rangle$
$\mathsf{V}_{h,K}$: $\langle -[\![\, h \,]\!]_K,\ +h\rangle$
$\mathsf{A}_{h,K}$: $\langle -K^{-1},\ -h,\ +[\![\, h \,]\!]_K\rangle$
H_h: $\langle -h,\ +\mathsf{hash}(h)\rangle$
TC_h: $\langle -h,\ +\mathsf{tag\ h}\rangle$
TS_h: $\langle -\mathsf{tag\ h},\ +h\rangle$

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