CAPSL Integrated Protocol Environment *

G. Denker and J. Millen
Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{denker || millen}@csl.sri.com

Abstract

CAPSL, a Common Authentication Protocol Specification Language, is a high-level language to support security analysis of cryptographic authentication and key distribution protocols. It is translated to CIL, an intermediate language expressing state transitions with term-rewriting rules. Connectors are being written to adapt CIL to supply input to different security analysis tools, including PVS for inductive verification and Maude for model-checking.

1. Introduction

It is generally understood that high-assurance network security relies upon the proper use of strong cryptography. It is also clear that a strong cryptosystem is ineffective without safe key management. In modern network environments, keys are distributed using connection establishment protocols, and applied in ways that are supposed to guarantee privacy or authentication of messages. Cryptographic protocols are the cornerstone of secure electronic banking and commerce, as well as military communications.

1.1. Protocol Vulnerabilities

It has been known, at least since the 1981 publication of a replay attack [DS81] on a key distribution protocol in [NS78], that cryptographic protocols are vulnerable to message modification attacks (see Figure 1). These attacks assume that an intruder can eavesdrop on communications and interfere with messages in transit. Message modification or "active" attacks include such tactics as message interception, replay, and address spoofing. Closely related activities on the Internet have been discovered. One notorious example is IP spoofing; the vulnerability was described by Bellovin in [Bel89], and in 1995 a hacker took advantage of it. This confirms the existence of sophisticated active attacks, though without the cryptographic aspect.

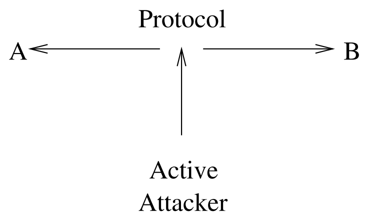

Figure 1. Message Modification Attacks

Some efficient methods have been developed to detect protocol vulnerabilities. Because of the logical, puzzle-like nature of this problem, the analysis is not performed on the executable code, but rather on an abstract model or "formal" specification of the protocol. Specifications of this kind treat cryptographic operations axiomatically, assuming that data cannot be decrypted without the proper key. This assumption distinguishes protocol failure analysis from other concerns, such as possible weaknesses in cryptographic algorithms. Protocol vulnerabilities, once detected, can usually be corrected by relatively small changes in the protocol.

1.2. Analysis Methods

There are three principal protocol analysis approaches: authentication logics, state-space exploration, and inductive proofs. (See Figure 2.) The first and last approaches aim at proving a protocol correct with respect to stated security properties, and the second searches for successful attacks. Notable successes of vulnerability analyses include an attack found with the NRL Protocol Analyzer on a selective broadcast protocol [Mea92] and an attack found with the FDR model-checker on the Needham-Schroeder public-key protocol [Low96]. We will not attempt to survey the history and status of these approaches here, but later on there are

* This work was supported by DARPA through the Air Force Research Laboratory under contract F30602-98-C-0258.

a few references to some of the work that has had a direct impact on the environment we are developing.

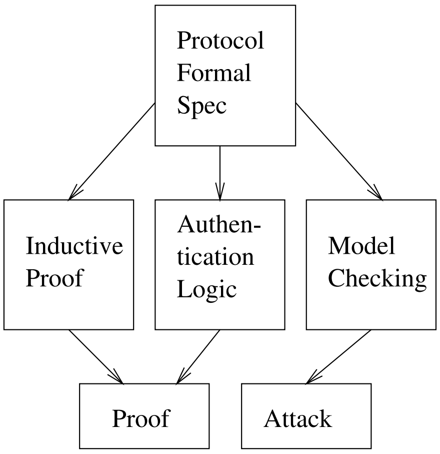

Figure 2. Protocol Analysis Methods

Most formal analysis techniques, even those originally intended to be used by hand, are supported by software tools. The tools require protocols to be expressed in a tool-specific language, with special conventions peculiar to cryptographic protocol analysis. The language may be low-level and based on the general-purpose language that the tool was implemented in or designed to support; or it may be a customized higher-level language. Several researchers realized that it was well worth the trouble to support a higher-level language, if only to save their own time in preparing protocols for analysis.

1.3. The CAPSL Integrated Environment

The acronym "CAPSL" stands for "Common Authentication Protocol Specification Language." CAPSL is a high-level language for cryptographic authentication and key distribution protocols. It is designed to be used as the input format for a variety of formal analysis techniques. The core of a CAPSL specification is a message list like the ones that are often used to present cryptographic protocols in articles and textbooks.

The idea of CAPSL as a single common protocol specification language that could be used as the input format for any formal analysis technique was first presented at the 1996 Isaac Newton Institute Programme on Computer Security, Cryptology, and Coding Theory at Cambridge University. This led to an initial version of CAPSL [Mil97] and later to the current effort.

The general architecture of the CAPSL Integrated Protocol Environment is illustrated in Figure 3. It shows the strategy by which CAPSL can be adapted for use by various protocol analysis tools. The core of this strategy is the use of an intermediate language, CIL (CAPSL Intermediate Language), that is closer to the state-transition representation used by almost all of the inductive proof and model-checking tools. Thus, CIL serves two purposes: to help define the semantics of CAPSL, and to act as an interface through which protocols specified in CAPSL can be analyzed using different tools. We envision customized sub-translators or "connectors" that translate from CIL to whatever form is required for each tool.

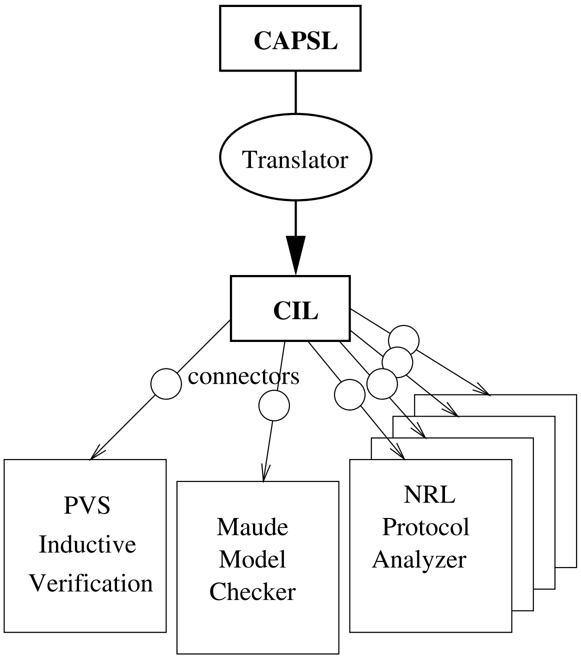

Figure 3. CAPSL Integrated Environment

The translator from CAPSL to CIL deals with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the state transitions of the communicating processes.

Translation of CIL to the input language of other tools is viewed as a task for researchers engaged in applying those tools to protocol analysis. At least two connectors will be constructed as part of our own efforts: for PVS and for Maude. PVS is the SRI verification environment, which we are using to support inductive protocol proofs. Maude is an efficiently executable specification language based on rewriting logic which has also been developed at SRI (see Section 5) and which is being used to support model checking. There is a CAPSL interface to the NRL Protocol Analyzer, but it does not use CIL [BMM99].

The integrated protocol environment is still under development. This paper offers an overview of the current state of the CAPSL and CIL design, translator design and implementation, and a snapshot of the current tool support.

2. CAPSL

CAPSL is modular and extensible, and has a number of syntactic features that are unique to protocol analysis. CAPSL syntax is motivated both by user convenience and by the needs of protocol analysis tools.

A CAPSL specification is made up of three kinds of modules: *typespec*, *protocol*, and *environment* specifications, usually in that order. Abstract data type specifications (called *typespecs*) introduce new data types and define cryptographic operators and other functions axiomatically. Standard typespecs are included automatically and others may be supplied by the user.

There may be more than one protocol specification, if there is a top-level protocol that invokes one or more subprotocols. Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools.

The following subsections take a closer look at protocol specifications, typespecs, and an environment. Some more advanced CAPSL features are also discussed.

2.1. Protocol Specification

Here is a small example of a protocol specification, illustrating several of the language features. It looks like a strongly typed program, with variable declarations. Unlike an ordinary computer program, the body is a sequence of protocol messages, and it has a list of security goals.

```
PROTOCOL OneMessage;
VARIABLES
    A, B: PKUser;
    K: Skey, FRESH, CRYPTO;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
    A -> B: {A,K}pk(B);
GOALS
    SECRET K;
    BELIEVES B: HOLDS A: K;
END;
```

This protocol has only one message, in which principal A sends a newly generated key K to another principal B. The key K is concatenated with A and encrypted using B's public key.

Note that this protocol can be instantiated many times with different principals as values for the protocol variables A and B. It helps to keep in mind the distinction between the principal itself, usually a named user (e.g., "Alice" or "Bob"), and the *role* it plays in the protocol. This protocol defines two roles, which could be described as initiator and responder. The name of the initiator role is "A" and the name of the responder role is "B." A principal Alice could play either the "A"-role (initiator) if it is the value of A, or the "B"-role (responder) if it is the value of B.

Protocol variables are typed, and they may also have *properties*. For example, the property FRESH means that the value of the variable is a new value generated for each session, and has not (with cryptographically high probability) been used before by the principal generating it. The property CRYPTO means that the value of the variable is unguessable (because it comes from a large domain and is chosen in an unpredictable way). Properties affect interpretation of the message list and also affect the attacker model.

The HOLDS assumption means that when an A process begins, it is initialized with the identity of a principal B to communicate with.

There are two security goals: that the new key K generated by A is kept secret from any attacker (unless the attacker happens to be either of the named users A or B), and that if the B session completes, B is justified in believing that A holds K.

The belief assertion means that `HOLDS A: K` is interpreted in the context of B's values for A and K. Thus, this is an agreement goal. It fails if there is no "A"-role process for this session with B's value for A and which holds the same value of K. For this protocol, the agreement goal could fail, since the message received by B, encrypted with B's public key, could have been forged by an attacker.

In general, assertions appearing as assumptions or goals are statements about protocol variables and the values held for them by the processes playing the various roles in a protocol session.

2.2. Type Specification

Messages are constructed using cryptographic operators and other functions, such as concatenation and hash functions. Every message field is of type Field, but certain operators require or produce fields of particular subtypes, such as key types.

A portion of the CAPSL type hierarchy is shown in Figure 4, including the type PKUser employed in the "OneMessage" protocol example. The fact that PKUser is a subtype of Principal means that a variable of type PKUser may be used wherever the type Principal is required, such as the source or destination of a message.

All functions used in CAPSL and the data types they operate on must be specified axiomatically in typespecs. The most commonly used types and operators are defined in a standard prelude. Analysts may add new types and operators by providing their own typespecs. Protocol specifications and other typespecs may import user-supplied typespecs.

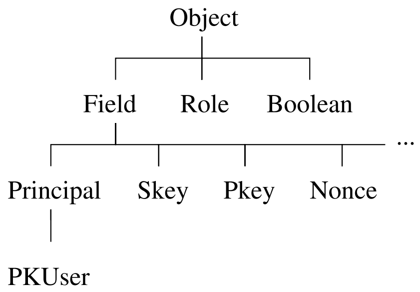

Figure 4. The Object Hierarchy

Here is an example of a typespec that defines the subtype PKUser of Principal. A PKUser has a public/private key pair. (The typespec PPK is actually part of the current prelude.)

```
TYPESPEC PPK;
TYPES PKUser: Principal;
FUNCTIONS
  pk(PKUser): Pkey;
  sk(PKUser): Pkey, PRIVATE, CRYPTO;
VARIABLES
  X: Field;
  U: PKUser;
AXIOMS
  {{X}pk(U)}sk(U) = X;
  {{X}sk(U)}pk(U) = X;
  INVERT {X}pk(U): X | sk(U);
  INVERT {X}sk(U): X | pk(U);
END;
```

PRIVATE Functions

The public key of a PKUser is found with the function `pk` and the corresponding private key with `sk`. The secret-key function is declared with the property PRIVATE to indicate that the function call `sk(U)` can be evaluated only by a process running on behalf of U. No other honest process, or the attacker, has direct access to `sk(U)`. This notion of private access is a novel feature of the CAPSL specification language; it is a different concept from the scope restrictions as found in Java and other languages.

Invertibility

The axioms for PKUser include two ordinary reduction axioms and two special invertibility axioms. An invertibility axiom is a special formula used by CAPSL to determine semantic correctness of messages. An axiom `INVERT` *term*: *var* | *termlist* says that a value given by *term* can be inverted to find the value of *var* provided that the value of each term in *termlist* is held. Sometimes invertibility can be inferred from reduction axioms, but sometimes it cannot.

A term like `{X}K` is invertible in the sense that a process holding `{X}K` and the inverse of K is able to extract X. Other terms have other rules for invertibility. The concatenation `[X,Y]` is invertible to extract either X or Y without any additional information. A bitwise exclusive-or `xor(X, Y)` is invertible to extract either X or Y given the other. If *hash* is a one-way function, `hash(P)` is invertible to extract P provided that the domain of the function is a type representing weak passwords (inversion is by guessing and comparing). The invertibility axiom is a uniform way to express all of these properties and any introduced by users.

Bracket Notation

Operators and functions in CAPSL are normally expressed in functional notation, e.g., `sk(Alice)`. However, curly and square brackets ({ } and [ ]) are commonly used in protocol descriptions to signify concatenation with encryption. In CAPSL, an expression like `{A, K}pk(B)` is shorthand for `ped(pk(B),cat(A,K))`. Here, `cat` is a concatenation operator which also can be used by itself (without encryption). The function `ped` is an RSA-like public-key encryption and decryption operator.

Square brackets identify another form of concatenation, with the operator `con`. The difference is that `cat` is associative, `con` is not. A term `{{A,B},C}` is equivalent to `{A,{B,C}}`. Thus, if a pattern $\{X, Y\}$ is matched to a message `{{A,B},C}`, X would be matched with `A`, while if the same pattern is matched to a message `[[A,B],C]`, X would get `[A,B]`.

When a bracket expression is followed by a key, as in `{A,K}Ks`, the encryption operator is assumed to be `ped` if `Ks` is of type Pkey, but it will be interpreted as the built-in symmetric-key encryption operator `se` if the key is of type Skey. If decryption rather than encryption is required, it is written `{A,K}'Ks`.

DENOTES

In protocol specifications, it is common to see names defined for significant expressions, which are then referred to by name in the message list. The DENOTES section in a CAPSL protocol specification can hold definitions like this. A kerberos ticket-granting ticket, for example, could be defined as

```
DENOTES Tgt = {Kct,C,Tgs,T}Ks
```

containing abstracted fields such as the client C, etc. An equation like the one for `Tgt` could have been inserted as an action in the message list. The DENOTES declaration simply features it in an easily found global location, to be referenced implicitly where needed. The symbol `Tgt` and the others are just protocol variables requiring their own type declarations as usual.

2.3. Environment Specifications

When a protocol is being analyzed by a model checker or simulated, the analyst may have to specify which processes are to be run. The analyst may also have to supply other run-specific information such as the initial knowledge of the attacker.

An environment specification defines constants of type Principal and some protocol sessions using them, and creates one or more *runs*, each of which is a scenario involving one or more of the sessions. A run is an interacting collection of sessions, something like a "bundle" in strand-space terminology [THG98]. A *session* is defined by allocating all the roles in a protocol specification to particular constants of type Principal.

For example, suppose we want a run with two sessions, one in which Alice takes the "A" role and Bob takes the "B" role, and another in which a dishonest principal Eve takes the "A" role and Alice takes the "B" role.

We might set that up like this:

```
ENVIRONMENT Test1;
IMPORTS OneMessage;
CONSTANTS
    Alice, Bob: PKUser;
    Eve: PKUser, EXPOSED;
SESSION S1;
    A = Alice;
    B = Bob;
SESSION S2;
    A = Eve;
    B = Alice;
RUN R1;
    EXPOSED sk(Alice);
    ORDER (S1||S2);
END;
```

Note that these session specifications only set up initial states of the processes in each role, and attacker action may prevent some of them from continuing or completing their normal programs.

Sessions are specified by associating roles with constants of type Principal. The role names are the variables of any subtype of Principal that occur in the imported protocol specification. Any other variables that are initially held by any principal must also be assigned in each session to some declared constant.

A principal (Eve, in this case) with the EXPOSED property yields all its PRIVATE values to the attacker. Eve may be thought of as colluding with, or subverted by, the attacker. For each run, there is also optionally a list of individual EXPOSED terms. (The attacker model is dicussed further below.)

The ORDER section can specify a series-parallel sequencing of sessions. Some search tools could benefit from this and save time when such a restriction is assumed. For example, if there were another session `S3`, we might say `ORDER ((S1;S2)||S3)` to mean that session `S2` does not start until `S1` ends, but that sequence runs concurrently with `S3`.

If no run information is given, CAPSL will assume that there is one run "Run1," with no additional terms exposed, in which all declared sessions are run concurrently.

2.4. Other CAPSL Language Features

View Notation

Sometimes a message is viewed differently by the sender and the receiver. The sender and receiver versions of the whole message, or of any individual fields, are separated by the symbol %. In the message `A -> B: {A,Ks}Kas%E` the encrypted term is received as a single value E if B will pass it on rather than decrypt it. This convention was originated by Lowe in Casper [Low98]. If the message were specified simply as `A -> B: {A,Ks}Kas`, it would mean that B was intended to decrypt or check the encrypted content, and CAPSL would report an error if B did not possess `Kas` or its inverse.

Actions

Messages may be preceded and followed by equational actions. An equational action like `X = {Y}K` may be either an assignment to X or a comparison test; it depends on whether or not the acting principal already holds a value for X. The semantics of a comparison, as reflected in the CIL rule output, is that if the test fails, the process cannot take any further state transition.

Interleaved Assertions

Besides their use as assumptions or goals, assertions may appear within a message list, like actions, between messages. An assertion in a message list is prefixed by the keyword ASSUME or PROVE to indicate its status.

There is presently no specific support for message idealizations used in authentication logics. GNY logic, in particular, permits terms in messages to be linked with *message extension* statements [GNY90]. We may add an operator like Brackin's || in ISL [Bra97] for this purpose.

Subprotocols and Selection

In larger protocols it is sometimes convenient to recognize functionally related groups of messages called "exchanges" or subprotocols. In CAPSL, a subprotocol is declared with an ordinary protocol specification, except that it imports a parent protocol to incorporate its variable declarations.

The parent protocol invokes the subprotocol by name where needed, using the form `INCLUDE` *name*.

Conditional selection of protocols is accomplished within a message list by putting `INCLUDE` invocations within an IF-THEN: `IF` *test* `THEN INCLUDE` *subprot1* `ELSE INCLUDE` *subprot2* `ENDIF`.

3. Interface Format For Analysis Tools

3.1. CIL

The challenge for the design of CIL is to make it easy for us and for other researchers to build the connectors needed to translate CIL into the input specifications used by the tools we want to use.

It must be expressive enough to represent a wide range of protocols, and yet at a low enough level to be close to the representation used by most verification or model-checking tools. Many such tools share a specification style that incorporates state-transition rules specified in a pattern-matching style, with symbolic terms to represent encryption and other computations. There is usually a separate and fairly standard attacker model, based on the concept in the early Dolev-Yao paper [DY83].

The meta-notation of Cervesato, et al [CDL$^+$99] was adapted for CIL because it appeared to be the simplest and most general formalism that exhibits these features. Their notation, according to the authors, could be regarded as either an extension of multiset rewriting with a kind of existential quantification, or a Horn fragment of linear logic. The logical style of this formalism made it suitable to serve as the language in which to express the semantics of CAPSL. Furthermore, the term-rewriting aspect corresponded well with the modeling approach taken by Denker, Meseguer, and Talcott with Maude [DMT98]. CIL may be regarded as a notational variant of the linear logic formalism in which certain specific conventions have been used to set up protocol models derived from CAPSL specifications.

3.2. Multiset Term Rewriting

The linear logic formalism uses transition rules of the form

$$F_1, ..., F_k \longrightarrow \exists X_1, ..., X_m : G_1, ..., G_n,$$

where each F_i and G_j is a "fact." Facts are atomic formulas of the form $P(t_1, ..., t_r)$ where P is a predicate symbol and the arguments t_i are terms. A term is constructed from typed constants, variables, and function symbols. Free variables are implicitly universally quantified.

The state of a system can be represented by a multiset of facts. (A multiset is a set in which there may be more than one copy of each element.) A rule is eligible to fire when the facts on the left side of the rule can be matched with facts in the multiset. When a rule fires, the matching facts in the multiset are removed from it and replaced by the facts on the right side of the rule, instantiated according to the substitution required by the pattern match. Removing a fact from the multiset reduces its multiplicity by one, if it was more than one. Facts in the multiset are typically ground terms (no variables) when finite-state search tools are used.

The existential quantifier in linear logic has a special meaning. Quantified variables are instantiated with fresh (unused) constants. This behavior is used to model generation of nonces.

Protocol State Facts

In protocol modeling, facts are used to express the entrance of a process into a state, the transmission of a message, or certain security-relevant events. In CIL, in particular, a state is represented by a fact `state`(*role, num, sid, termlist*) where a role is a constant corresponding to a principal, *num* is a state label, usually an integer, *sid* is a unique session identifier, and the terms in the *termlist* are the "memory" of the process in that role and state. A message is a fact `msg`(*sender, receiver, sid, termlist*). A `net`(*term*) fact represents attacker knowledge.

Rules with an empty left side are interpreted as initialization or fact-generating rules. For each role in the protocol, an initial state fact is generated with initially held variables. The rule

$$\begin{aligned}\rightarrow \exists i : &\ \mathtt{state}(\mathtt{roleA}, 0, i, A, B),\\ &\ \mathtt{state}(\mathtt{roleB}, 0, i, B)\end{aligned}$$

creates two facts representing the initial state of two new processes of the protocol, one in role "A" and the other in role "B". The role names are constants taken from the text of the protocol specification, and should be distinguished from the variables A and B, which are free variables whose names could be changed without changing the meaning of the rule. Since A and B are variables of type Principal, this rule can initiate sessions between any pair of principals. Each newly initiated session has its own unique session identifier i.

The message `A -> B: {N}sk(A)` would result in two transitions, one for the sender A and one for the receiver B. Assuming that A is in state 0 and N is a nonce originating in this message, the A transition would be:

$$\begin{aligned}&\mathtt{state}(\mathtt{roleA}, 0, i, A, B) \rightarrow\\ &\exists N : \mathtt{state}(\mathtt{roleA}, 1, i, A, B, N),\\ &\qquad \mathtt{msg}(A, B, i, \mathtt{ped}(\mathtt{sk}(A), N)).\end{aligned}$$

Attacker Facts and Rules

The Dolev-Yao model of the attacker, commonly used in protocol analysis, can be expressed as a set of rules that allow the attacker to eavesdrop on transmitted messages and to forge new messages from available message components.

In principle, the attacker model is very simple. For each message fact $\mathtt{msg}(A, B, i, X)$, there is a rule

$$\mathtt{msg}(A, B, i, X) \rightarrow \mathtt{net}(X), \mathtt{msg}(A, B, i, X)$$

which generates a fact $\mathtt{net}(X)$ asserting that the attacker now holds the content of the message. (Note that the message is repeated on the right so that it will still be available in the multiset for the intended receiver after the attacker reads it.) There are also rules giving the attacker any terms declared as EXPOSED in the environment.

Other attacker rules allow the attacker to analyze complex terms into their components, for example

$$\mathtt{net}(\mathtt{se}(K, X)), \mathtt{net}(K) \rightarrow$$
$$\mathtt{net}(X), \mathtt{net}(\mathtt{se}(K, X)), \mathtt{net}(K)$$

The attacker can also construct complex terms from the ones it holds, for example

$$\mathtt{net}(X), \mathtt{net}(Y) \rightarrow$$
$$\mathtt{net}(\mathtt{cat}(X, Y)), \mathtt{net}(X), \mathtt{net}(Y)$$

and to put any term it knows into a message:

$$\mathtt{net}(X) \rightarrow \mathtt{msg}(\mathtt{Spy}, A, i, X), \mathtt{net}(X)$$

In this message, `Spy` is a constant principal identifying the attacker. Spy is the sender because we have adopted Paulson's convention that the message source and destination arguments are the true source and intended destination of the message, and they are not directly accessible to the receiver of the message. The message is "forged" in the sense that the message content has been constructed by the attacker, and the message can be directed to any principal as its destination. More generally, we could have a predicate `spy( )`, and any principal E such that `spy(E)` is true would have the attacker's properties. If a constant `Eve` is declared as EXPOSED, a fact `spy(Eve)` is generated.

These rules can be extended to deal with messages having several fields.

This attacker model is too simplistic to be useful for model checking. It is noted in [CDL+99], for example, that it is helpful to separate decomposition from composition transitions by using two kinds of facts. However, while every protocol analysis tool needs an attacker model, most of the attacker model is fixed, and protocol-dependent aspects of it are influenced as much by the needs of the tool as by the details of the protocol. For this reason, it does not appear necessary or appropriate for the CIL translator to generate the attacker facts and rules. Our definition of the attacker is needed only to clarify the semantics of the CIL protocol environment, so that analysts can check that their attacker model is consistent with CIL.

3.3. Security Goals

Security goals in CAPSL include SECRET assertions and agreement assertions, expressed as beliefs about HOLDS statements. Other security goals are possible, but we are still in the process of working out their semantics.

The semantics of a security goal consists in an understanding of when it has failed. A secrecy goal has failed when the attacker has learned a value declared to be secret. Since the attacker can always transmit any value it learns in a message, we can recognize a secrecy failure when a secret value appears as the content of a message. That is, if X is secret, the presence of a message fact $\mathtt{msg}(P, Q, i, X)$ is a secrecy failure, for any principals P and Q.

But how do we know which values are secret? In the multiset model, a value X is declared as secret by generating a fact that says so. If the secrecy goal is `SECRET X: A,B`, the secrecy fact might be of the form $\mathtt{secret}(X, i, \{A, B\})$. This fact must be generated from some rule whose left side identifies X, i, A, and B from some state fact. This is most conveniently done as an extension to the rule that first transmits X in a message. Note that if A or B is the Spy, release of the secret X is not considered a failure, since X was intended to be shared with the Spy.

The translator could perform the service of adding secret facts to the right side of rules that generate secrets. At present it does not; it merely passes the SECRET declaration on to the connector.

An agreement goal `BELIEVES B: HOLDS A: X` is violated if B reaches its final state $\mathtt{state}(\mathtt{roleB}, n, i, B, A, X')$ with a value in the X position that differs from what A actually holds, as indicated by another state $\mathtt{state}(\mathtt{roleA}, m, i, A, B, X)$ where $X \neq X'$. (It can also fail if there is no corresponding state for `roleA`.) The use of the session identifier is essential here, since it does not matter if B disagrees with A from some other session!

3.4. Translator Output

A connector from CIL to any analysis tool needs not only the transition rules, but also all the other information supplied in the specification: variable and function declarations, assumptions, goals, and environment. The translator output has all of this in a digested form.

The output of the overall translation is:

1. symbol table and slot table
2. axioms (collected from typespecs, including the prelude, and environments)

3. localized assumptions
4. protocol rewrite rules, i.e., rules for state transitions
5. localized goals
6. environment information

The symbol table lists the types of variables and the type signatures of functions, with entries of the form:

`symbol`(*ident, status, args, val, props*)

where *ident* is the name of the variable, constant, or function, *status* is *var* for dummy variable (from an axiom), *pvar* for a protocol variable, *op* for a constant or function name, or *type* for a type name. The *args* entry is a list of the argument types (empty for a variable or constant), and *val* is the value type. Finally, *props* is a list of the property keywords that were given in the original CAPSL declaration.

The slot table relates state components with protocol variables. The fact that the value held for variable A is the first component in the state of `roleA` (after the required items of the role, state number, and session ID) is indicated by the slot table entry `slot(A,roleA,1)`. State component positions depend only on the role and variable and not on the state number. The reason that we need slot information is to interpret assumptions and goals, which mention protocol variables, in terms of the state facts.

Axioms from typespecs and environment specifications are consolidated into a single list.

Assertions are *localized* by associating them with states, inidicating the role and state number. In particular, assumptions are associated with the initial state, goals are associated with the final state, and interleaved ASSUME and PROVE assertions are associated with the current state where they are placed.

Currently we carry over environment information without change from the CAPSL, except that constant declarations are extracted and incorporated into the single collected set of declarations and that the axioms from the environment specification are consolidated with typespec axioms into a single list.

The form of CIL output is defined with an "abstract syntax" grammar that could be concretized in various ways. Presently, the output is a text file containing the elements described above in a functional notation. There is a technical report with a preliminary description of CIL output [DM99a]; current documentation on the details of CIL output may be found at the TIPE/CAPSL Web site [Mil99a].

Example CIL Output

For the sake of example, and for use in subsequent sections, here is a sample of part of the CIL output derived from the CAPSL modules OneMessage and PPK. The output normally includes additional material from the prelude.

```
symbols(
  symbol(PKUser,type,ids(),Principal,
         props()),
  symbol(pk,op,ids(PKUser),PKey,props()),
  symbol(sk,op,ids(PKUser),PKey,
         props(PRIVATE,CRYPTO)),
  symbol(X,var,ids(),Field,props()),
  symbol(U,var,ids(),PKUser,props()),
  symbol(A,pvar,ids(),PKUser,props()),
  symbol(B,pvar,ids(),PKUser,props()),
  symbol(roleA,op,ids(),Role,props()),
  symbol(roleB,op,ids(),Role,props()),
  ...)
axioms(
  eqn(ped(sk(U),ped(pk(U),X)),X),
  eqn(ped(pk(U),ped(sk(U),X)),X),
  invertible(ped(pk(U),X),X,terms(sk(U))),
  invertible(ped(sk(U),X),X,terms(pk(U))),
  ...)
slots(
  slot(A,roleA,1),
  slot(B,roleA,2),
  slot(B,roleB,1),
  slot(K,roleA,3),
  slot(A,roleB,2)
)
rules(
  rule(
    facts(),
    ids(SID),
    facts(state(roleA,0,SID,terms(A,B)),
          state(roleB,0,SID,terms(B)))),
  rule(
    facts(state(roleA,0,SID,terms(A,B))),
    ids(K),
    facts(state(roleA,1,SID,terms(A,B,K)),
          msg(A,B,SID,
              terms(ped(pk(B),cat(A,K)))))),
  rule(
    facts(state(roleB,0,SID,terms(B)),
          msg(UNK,B,SID,
              terms(ped(pk(B),cat(A,K))))),
    ids(),
    facts(state(roleB,1,SID,
                terms(B,A,cat(A,K)))))
)
goals(
  loc(
    nodes(node(roleA,1),node(roleB,1)),
    secret(K,ids())),
  loc(
    nodes(node(roleA,1),node(roleB,1)),
    believes(B,holds(A,ids(K))))
)
```

In the last rule, where B receives the message, the sender of the message is designated with a free variable `UNK` that does not have to match A. This is because we have adopted

Paulson's convention that the message source and destination arguments are the true source and intended destination of the message, and they are not directly accessible to the receiver of the message. Hence B will accept any message of the given form regardless of the true sender, and will learn values for only those protocol variables that occur in the message content.

3.5. Translator Implementation

The CAPSL-to-CIL translator is presently implemented partly in Java and partly in Maude. The Java portion of the translator parses the CAPSL, performs type checking, and consolidates declarations, axioms, and assumptions. It flags syntax errors or type consistency violations. If there are none, it produces a text file containing the symbol and slot tables, assumptions, goals, message list and environment.

The Maude portion of the translator converts the message list into transition rules and localizes assertions. It checks that the message list is *implementable* in the sense described in the next section. Essentially, it determines that a process is capable of constructing messages it is supposed to send. This check is analogous to what a compiler does to check that program variables have been initialized before they are used.

The translator can be run with a simple script that calls Java on the main class for type checker and pipes the output to the Maude runtime executable, which interprets a Maude rule-generation program. We plan to re-implement the rule generation in Java, so that the entire translation to CIL can be completed within Java.

4. Translator Algorithm

While the transition-rule part of CIL has its own rewriting-logic semantics, the semantics of CAPSL lies in its translation to CIL. In this section we summarize the main concepts of the message-list translation. More details can be found in [DM99b].

The translator goes through the messages and actions in sequence. Each message or action produces as output a pair of CIL transition rules, and it may also cause slot table entries to be added. In this way the translator keeps track of which variables are held by each role at each point in the protocol.

4.1. Computability and Receivability

While translating messages into rules, the translator checks the implementability of the protocol. Different checks are made depending on what a protocol process has to do, but the key concepts are computability, receivability, and invertibility.

As we noted in the general context of multiset term rewriting, a message causes two state transitions: one for the sender, which generates a rule with a message fact on the right as well as a new state, and one for the receiver, which consumes a message fact on the left.

A equational action can be either an assignment or a comparison. An assignment causes a program variable to become held, so it adds a new role state component, creates a slot table entry, and puts the term on the right into the state. This state change is reflected in a CIL rule. A comparison similarly adds a role state component containing the equation. Then it adds another transition rule to the CIL output which fires when that component contains the value "true."

When the translator creates a rule for the sender, it checks that the content of the message to be sent is *computable*. A term is computable if it can be constructed from the sender's state components, using declared functions if necessary. Functions declared PRIVATE can be employed only when the principal named in the first argument is the sender.

When the translator creates a rule for the receiver, it checks that the content of the message to be received is *receivable*. A term is receivable if it is computable, or if it is a protocol variable that is not yet held, so that receiving the message causes the receiver to add that variable to its state. In that case the translator creates a new slot table entry for that variable and the receiver.

If a term is not computable, it is receivable if it is invertible, and its components are (recursively) receivable. Invertibility is checked using the INVERT declarations. If invertibility requires that some term be held, such as an inverse key, that term must be computable.

Similar considerations apply in the processing of equational actions.

Rule Optimization

The translator produces two rules for each message. Often, successive transitions for the same role can be combined into a single transition.

Consider the two messages

```
A → B: A;
B → A: B;
```

Two transition rules are generated for B:

$\mathtt{state}(\mathtt{roleB}, 0, i, B), \mathtt{msg}(A, B, i, A) \rightarrow$
$\mathtt{state}(\mathtt{roleB}, 1, i, B, A)$

$\mathtt{state}(\mathtt{roleB}, 1, i, B, A) \rightarrow$
$\mathtt{state}(\mathtt{roleB}, 2, i, B, A), \mathtt{msg}(B, A, i, B)$

These can be combined into the rule

$$\texttt{state}(\texttt{roleB}, 0, i, B), \texttt{msg}(A, B, i, A) \rightarrow$$
$$\texttt{state}(\texttt{roleB}, 2, i, B, A), \texttt{msg}(B, A, i, B)$$

This does not eliminate any possible attacks, since nothing any other process could do between B's first transition and B's second transition can affect B; and any transitions taken by other processes before B sent its reply do not depend on B's state, and could still be taken if B's reply is earlier.

Optimizing the rule set is an advantage for verification or model-checking. Fewer rules means fewer transitions, which means a smaller search space. The performance of a finite-state exploration tool can be increased enormously in this way.

5. Analysis of Cryptographic Protocols

Given the CIL output, the next step is to develop a connector to translate the CIL output into a form suitable for analysis with some existing tool or technique. We have been doing protocol analysis using PVS for inductive proofs and Maude for model checking, and connectors for translating a CIL protocol specification into these environments are under development.

This section has some examples constructed by hand of how the translation would work. The connector algorithms are still changing, because some details of the CIL output and also some details of the analysis techniques are undergoing revisions. However, both analysis techniques have been used successfully, and the principal features of the connectors are well established.

5.1. Connector: PVS Inductive Verification

Inductive verification of protocols is the only known way to prove a protocol correct that is deep enough to prove secrecy properties. Our approach uses a trace model based on work by Paulson [Pau98]. To be useful for inductive proof of protocols, a specification and theorem-proving environment should support abstract data types and convenient induction. PVS is such an environment, based on higher-order logic and augmented with a library of decision procedures and the ability to add user-defined proof strategies [ORS92].

A trace model differs from the multiset model because all message facts (called "events" in the trace model) are saved in order, in the trace, instead of being deleted when they are received. A trace model may also save state events. It is easy to reinterpret a CIL state transition rule as a rule for updating the trace.

Before giving an example of how a CIL rule becomes a trace update, it is necessary to provide some background on the structure of a protocol specification in PVS.

PVS Protocol Specification

The PVS specification has some protocol-independent supporting "theories" that correspond roughly to the typespecs in the CAPSL prelude. Theories contain both data type definitions and lemmas about those data types. There is a theory defining traces and messages, with different types of message fields such as keys and principals. Other theories import this one, such as the theory that defines Paulson's set operators *analz* and *synth* and contains lemmas giving their properties. This theory also defines the attacker's ability to construct a forged message from data extracted from a trace. Built on these is a theory defining the strand-space *ideal* from [THG98] and the related notion of a *coideal* defined and used in our own work, such as the proof that a certain protocol has only a parallel attack [Mil99b].

The protocol itself is defined as a set of acceptable traces. The definition is recursive: the empty trace is acceptable, and a new message event or state event can be added to an existing trace if it satisfies some state-transition rule. Thus, transition rules become predicates with two arguments, an event and a trace. The protocol as whole is defined as a global predicate that adds an event to a trace if it satisfies any one of the protocol-rule predicates, or an attacker-rule predicate.

For example, consider the rule:

```
rule(facts(state(roleB,0,SID,terms(B)),
           msg(A,B,SID,terms(A))),
     ids(),
     facts(state(roleB,1,SID,terms(B,A)),
           msg(B,A,SID,terms(B))))
```

This is transformed into a predicate

```
B0(E,T): bool =
  EXISTS (SID,B,A):
   (member(State(roleB,0,SID,B),T)
    AND member(Msg(A,B,SID,A),T)
    AND E = State(roleB,2,SID,Con(B,A))
    ) OR
   (member(State(roleB,1,SID,Con(B,A)),T)
    AND E = Msg(B,A,SID,B))
```

Without going into the details of the PVS syntax, it should be easy to see the close relationship between the CIL and the PVS predicate. This is what makes the connector to PVS easy to write. The complete trace predicate is a disjunction:

```
Protocol(E,T): bool =
  A0(E,T) OR B0(E,T) OR A1(E,T) OR ...
  OR Fake(E,T)
```

where the "Fake" predicate is from a fixed attacker model.

There is an apparently significant difference between the trace model and the multiset model: because old messages and states remain in the trace, a predicate like the one above can be invoked repeatedly to add any number of copies of the same new state and message. From a security verification point of view, this is not a problem, since any attack on the "real" protocol is still possible in this more liberal model. Conceivably one could find an attack in this model that was not real, because it could send information from new message to an old state. Spurious attacks like this are easy to recognize, and they could also be eliminated by modifying the transition rules by putting in an additional test.

Type Declarations

A full protocol theory includes not just the trace predicate but the type declarations of the variables used in it. For this example, that would include

```
E: VAR event
T: VAR trace
A, B: VAR principal
SID: VAR sessid
```

and others. These are easily obtained from the CIL symbol table output. The declarations of the types are in the standard supporting theories, so that the prelude-generated part of the symbol table is ignored by the PVS connector.

At present, we are able to deal with encryption and key-generating functions only if they are already provided by the existing PVS supporting theories. However, there is no reason why we cannot extend support in the future to user-defined functions given in typespecs, since PVS has quite general abstract datatype specification capabilities.

Goals

A security goal, for purposes of inductive proof, is a trace invariant. The general comments about security goals in Section 3.3 apply. A secrecy goal says that if a term X is secret, no message event has X as its content. A term is a secret if an event `secret(X,SID,terms(A,B))` declaring it as such appears in the trace. This is another kind of event, and in the PVS specification, the protocol predicate must be augmented with another rule that generates it. Generally, the state transition that first introduces X in message is the one that is given the new `secret` event.

At present, the PVS connector must look for `secret` goals in the CIL output and use the slot table information to locate the appropriate transition rule.

Agreement goals are also invariants: if the current state event is the last state of B, and B is supposed to agree with A on K, the trace is tested for the presence of an A-state with the same K in the same session. No additional special events are needed.

5.2. Connector: Maude Model Checker

We have had encouraging experiences with Maude [CELM96] for the specification and analysis of cryptographic protocols. The level of specification is detailed enough to capture essential operational aspects. The reflective capabilities of rewriting logic [Mes92, Mes96b] and Maude [CM96] are very helpful, because they support various analysis techniques such as state search, theorem proving, and symbolic simulation. Figure 5 illustrates how this is done in Maude.

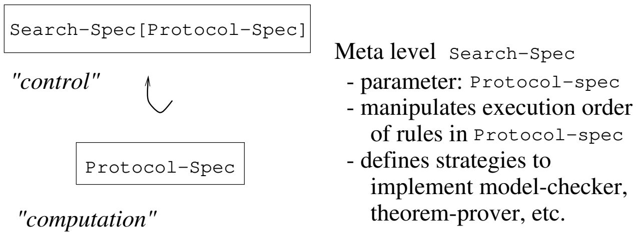

Figure 5. Reflection in Maude

The meta-level specification `Search-Spec` takes `Protocol-Spec` which defines the behavior of the cryptographic protocol as a parameter. At the meta-level the execution of transition rules of the protocol is controlled via user-defined execution strategies. A breadth-first search strategy can exhaustively explore all the executions of a system. Under finite-state or other termination assumptions it is even possible to analyze all executions. This is very helpful in uncovering security flaws under unforeseen attack scenarios.

Maude Basics

There are two types of modules used in Maude to specify cryptographic protocols: *functional* and *system* modules. Functional modules are equational theories used to specify algebraic data types. Cryptographic operators together with their axiomatic behavior can be represented in functional modules. System modules are rewrite theories specifying concurrent systems. Rewrite rules are used in these modules to express the dynamics of a system. All the above modules can be parametrized. This allows vertical and horizontal structuring.

Modeling Protocols in Maude

The CIL specification of a cryptographic protocol is the starting point to derive the corresponding Maude executable specification. The symbol table declarations and the axioms of CIL are directly translated into abstract data type specifications in Maude, protocol rewrite rules correspond to

Maude rewrite rules, and environment information is used to declare test scenarios for which the protocol can be executed and analyzed. Figure 6 summarizes the translation.

CIL symbol table declaration	$\longrightarrow$	Maude declarations
type	$\dashrightarrow$	sort
subtype	$\dashrightarrow$	subsort
op	$\dashrightarrow$	op
pvar	$\dashrightarrow$	var
Axioms	$\longrightarrow$	Equations
Protocol Rewrite Rules	$\longrightarrow$	Rewrite Rules
Environment Spec	$\longrightarrow$	Initial configurations

Figure 6. CIL - Maude

The CIL declarations of the OneMessage example translate into the following Maude functional module.

```
fmod USER is
  including PRELUDE .
  sort PKUser .
  subsort PKUser < Principal .

  op pk : PKUser -> PKey .
  op sk : PKUser -> PKey .

  var X : Field .
  var U : PKUser.

  eq ped(sk(U), ped(pk(U), X)) = X .
  eq ped(pk(U), ped(sk(U), X)) = X .
  eq invertible(ped(pk(U),X),X,terms(sk(U)))
     = true .
  eq invertible(ped(sk(U),X),X,terms(pk(U)))
     = true .
```

Here, we are assuming that `PRELUDE` is a functional module in which predefined cryptographic types (e.g., `Field`, `Principal`, `PKey`), operators (e.g., `ped`) and predicates (`invertible`) are declared.

In Section 3.4 we presented the CIL rules for the OneMessage protocol. The translation into Maude rewrite rules is very straightforward. For instance, given Maude declarations for state, msg, and fact predicates, the sender CIL rule becomes the Maude rule

```
rl [send] :
facts(state(roleA,0,SID,terms(A,B)), FACTS)
=>
facts(
  state(roleA,1,SID,terms(A,B,mkKey(FACTS))),
  msg(A,B,SID,
      terms(ped(pk(B),cat(A,mkKey(FACTS))))),
  FACTS) .
```

`FACTS` is a variable of type multiset of facts which for a given system configuration is matched with all the remaining facts. `mkKey: FactMultiSet -> PKey` creates a fresh key from a given set of facts. Axiomatically `mkKey` is defined in such a way that the freshness property is guaranteed. In our current implementation we use natural numbers for key creation and we make sure that at any time while executing a specific protocol run (i.e., an ordered set of sessions), no two values of fresh variables are the same.

The Attacker Model in Maude

The following Maude rules describe an attacker that can overhear messages and create fake messages. We assume that principals and session identifiers that are used in scenarios are made available to the attacker during initialization.

```
rl [overhear] :
   facts(msg(A, B, SID, TERMS), FACTS)
   => facts(msg(A, B, SID, TERMS),
            net(TERMS), FACTS) .

rl [fake] :
    facts(net(A), net(SID), net(TERMS),
          FACTS)
    => facts(net(A), net(SID), net(TERMS),
             msg(Spy, A, SID, TERMS),
             FACTS) .
```

This attacker has the ability to analyze messages via decomposition and decryption. Decomposition is done in one step with a computation like Paulson's *analz* operator [Pau98].

```
rl [decompose] :
   facts(FACTS)
   => facts(closure(unCons(nets(FACTS)),
                    deCrypt(nets(FACTS))),
            noNets(FACTS)) .
```

`nets` and `noNets` are projection functions which split a given multiset of facts into net-facts and other facts. `unCons` and `deCrypt` are one-step decomposition and decryption operators, respectively. That means, given a multiset of facts, all lists are taken apart into their sublists as well as all encrypted fields are decrypted if a corresponding key is part of the attacker knowledge. The `closure` function recursively applies both these functions until no more new fields are gained.

Whereas decomposition of a given set of attacker facts is a terminating task, composition and encryption are not. The attacker may apply arbitrarily many layers of concatenation and encryption in order to construct a message which might result in a successful attack. This makes the search space for a given run of the protocol infinite, even though the number of honest agents and attackers may be finite. Thinking in terms of exhaustive search strategies, the resulting search tree is an infinite data structure. The root of this tree corresponds to the given initial protocol configuration (as given in the environment specification) and each tree node represents one possible extension of the parent node where a transition has taken place. Since we want to make a state-space search feasible, we chose to build a finitely branching search tree. The number of successors of each node is finite, but the tree is arbitrarily deep. For this reason, we chose to implement the attacker in such a way that only one encryption or one composition is allowed per step. Nested encryption and composition is achieved by applying a combination of the composition rules in sequence.

```
rl [composeCons] :
   facts(FACTS, net(FIELD), net(TERMS))
   => facts(FACTS, net(FIELD), net(TERMS),
                   net(FIELD,TERMS)) .

rl [composeEncrypt] :
   facts(FACTS, net(PKEY), net(TERMS))
   => facts(FACTS, net(PKEY), net(TERMS),
                   net(ped(PKEY,TERMS))) .
```

Defining an attack

In order to successfully detect protocol vulnerabilities it is necessary to not only build an abstract model of the protocol and the malicious environment which is amenable to formal analysis techniques, but also to give a formal definition of what constitutes an attack. In other words, when has a protocol security goal failed?

For instance, the secrecy goal in OneMessage can be expressed by introducing a `secret` predicate which in its first parameter states the variable which should be kept secret, and in its second predicate lists the protocol variables which share this secret. An attack situation is characterized by a situation where the secret is known to the attacker.

```
ceq facts(secret(K,terms(A,B)),net(K),FACTS)
    = attack
    if not(spyOf(A) == true)
       and not(spyOf(B) == true) .
```

The appropriate `secret` predicate has to be created along with other facts. As soon as a secret is created or is received by an agent, the secret predicate is created. This can be done by extending the original CIL rules with further predicates. For instance, at the same time that the agent in role "A" in OneMessage creates the secret key in order to send it to the agent in role "B" a corresponding secrecy predicate is created.

```
rule(
  facts(state(roleA,0,SID,terms(A,B))),
  ids(K),
  facts(state(roleA,1,SID,terms(A,B,K)),
        msg(A,B,SID,ped(pk(B),cat(A,K))),
        secret(K,terms(A,B))))
```

Model Checking Via Exhaustive Search Strategies

Our systematic search for an attack using a breadth-first strategy illustrates a form of model checking in which we explore all the possible behaviors beginning with a given state using the reflective capabilities of Maude. Thanks to the reflective nature of rewriting logic [CM96], it is possible to express strategies for executing Maude specifications in Maude itself. This is particularly important because Maude specifications formalize concurrent systems that can be highly concurrent and nondeterministic, and that may not terminate. Therefore, in addition to a default execution strategy that ensures a fair application of the rewrite rules, it is possible to define a very wide variety of rewriting strategies using rewrite rules at the metalevel, and to then execute a specification with a given strategy in Maude. For example, one can program a breadth-first-search strategy to exhaustively determine all possible behaviors from a given initial state using the rewrite rules of a given protocol specification. In this way we can either validate the specification and detect errors, or do formal reasoning about the specification.

To efficiently implement a breadth-first-search strategy, we have chosen to use a bounded depth-first-search strategy which is recursively called with increasing bounds as long as no attack is discovered. The search strategy is defined at the meta-level as a module which includes the protocol as a parameter.

```
fmod SEARCH is
  protecting META-LEVEL[OneMessage] .

  op bdfs : Module Term MachineInt
            LabelList -> Strategy .

  eq bdfs(MODULE,TERM,N,LIST) = ...
```

The bounded depth-first-search operation is equationally specified. The operation is called with the name of the protocol, an initial configuration (several of which are defined in the environment specification), the maximal search depth and a list of rule labels (i.e., all the names of rewrite rules over which the search strategy is defined). The bounded depth-search-first algorithm continues building the path, provided the maximum depth is not reached. For each new step in the path all rewrite rules are tried. Whenever a path is closed, i.e., whenever no other rewrite rules are applicable, or the maximum depth is reached, the algorithm backtracks. Backtracking means that first the same rule as in the previous step is tested with the next possible matching. If this fails, then the next rule in the list of applicable rules is tried. If this also fails, another backtracking step is taken until no more rules can be applied from the initial configuration. If any of the previous two attempts is successful, the bounded depth-first-search algorithm for the new, extended path is called.

After each step in the search it is tested whether an attack situation occurred. In the case where the violation of a goal is defined via an equation, as for instance the secrecy goal of OneMessage, this test is an equality, since the Maude rewrite engine automatically applies all equations after a rewrite step in order to derive a normal form.

Enhancement of Search Strategies

The search strategies we implemented are designed for reusability. They can be applied to arbitrary protocol specifications. So can the attacker model. The drawback of this generality is poor performance. Model checking through exhaustive state search easily reaches its limits even for small protocols and very limited numbers of participating processes. Therefore, it is highly desirable to enhance the search techniques. We have been experimenting with several possibilities for more efficient, but still complete, search algorithms. Our enhancement efforts can be divided into two main approaches: reordering and pruning of the search tree.

We reordered the search tree to give preference to actions which are more likely to result in an attack. This has no effect on the execution time unless the protocol has flaws, in which case it terminates more quickly. We gave priority to those rules which were generated by the protocol message list. This is due to the fact that their application is very restricted. Attacker rules are more generally applicable, thus they have lower priority. Among those rules, we prefer to apply fake messages over overhearing, and we assigned the lowest priority to composition of messages.

Pruning the search tree cuts down the number of states to be checked for a possible attack. One way to prune the search tree is to test faked messages for their format and receivability. The idea is that only messages which correspond to a valid protocol message are meaningful. Since we are working in a strongly-typed system, we only allow the attacker to fake a message that has the type of one of the messages in the message list. Additionally, a faked message should have an effect on the participating processes. That means that at least one process should be in a state where the faked message could be received, provoking this process to perform the specified transition. We call this kind of pruning "Send-Receive-Dependency".

6. Conclusion

CAPSL, CIL and the translation between them are designed to address important goals in cryptographic protocol specification for analysis purposes. With a common specification language, it becomes possible to harness the combined power of many tools for protocol analysis in a practical way. The components of the CAPSL environment include mobile software for translation of CAPSL to CIL, and connectors to adapt CIL to the input languages of various analysis tools. This software is still under development, but a preliminary CAPSL-to-CIL translator is already available on the Web site [Mil99a]. The site also contains more extensive documentation on CAPSL, including examples of CAPSL specifications, the full grammar, and also abstract syntax specifications for CAPSL and CIL.

With CAPSL, one can express protocols in the simplest accepted message-list form. Type specifications in CAPSL and their use for introducing new operators and subtypes bring an expanding class of protocols within reach. There are plans to broaden the applicability of CAPSL further with extensions for multicast protocols.

CAPSL simplifies what used to be the most awkward aspect of abstract protocol specification, the distinction between short-term session data and the long-term data associated with persistent entities. It was done by applying the general type specification mechanism, together with the novel concepts of private functions and invertibility axioms.

The intermediate language CIL was chosen with an eye toward a clear analysis-level modeling semantics and a universal pattern-matching transition rule style that lends itself both to model checking and inductive proof techniques. One novel modeling convention in CIL is the use of a session identifier, which facilitates the expression of agreement goals, and captures a protocol implementation's ability to distinguish between different sessions.

We are developing out techniques for inductive protocol proof using PVS and model checking using Maude. In the process, we have confirmed that CIL output is a good match for the specification needs of these tools, and we are in a position to finish up the connectors when those techniques and the CIL translator have stabilized.

References

[Bel89] S. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[BMM99] S Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET '99)*, 1999.

[Bra97] S. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Symposium on Network and Distributed System Security*. Internet Society, February 1997.

[CDL$^+$99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th IEEE Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society, 1999.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [Mes96a], pages 65–89.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In Meseguer [Mes96a], pages 125–147.

[DM99a] G. Denker and J. Millen. CAPSL and CIL language design. Technical Report SRI-CSL-99-02, SRI International Computer Science Laboratory, 1999.

[DM99b] G. Denker and J. Millen. CAPSL intermediate language. In *FLoC Workshop on Formal Methods and Security Protocols*, 1999.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.

[DS81] D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8), August 1981.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983. Also STAN-CS-81-854, May 1981, Stanford U.

[GNY90] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[Low98] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[Mea92] C. Meadows. Aplying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes96a] J. Meseguer, editor. *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996*. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

[Mes96b] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report. In U. Montanari and V. Sassone, editors, *Proc. 7th Intern. Conf. on Concurrency Theory: CONCUR'96, Pisa, August 1996*, pages 331–372, 1996. LNCS 1119.

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[Mil99a] J. Millen. CAPSL Web Site. `http://www.csl.sri.com/~millen/capsl`, 1999.

[Mil99b] J. Millen. A necessarily parallel attack. In *FLoC Workshop on Formal Methods and Security Protocols*, 1999.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–998, December 1978.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Pau98] L Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[THG98] J. Thayer, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *11th IEEE Computer Security Foundations Workshop*, pages 66–78. IEEE Computer Society, 1998.