CAPSL Interface for the NRL Protocol Analyzer

Stephen Brackin
Arca Systems
Ithaca, NY
brackin@arca.com

Catherine Meadows
Code 5543
Naval Research Laboratory
Washington, DC 20375
meadows@itd.nrl.navy.mil

Jonathan Millen
SRI International
Menlo Park, CA 94025
millen@csl.sri.com

Abstract

The Common Authentication Protocol Specification Language (CAPSL) is a high-level language for applying formal methods to the security analysis of cryptographic protocols. Its goal is to permit a protocol to be specified once in a form that is usable as an interface to any type of analysis tool or technique, given appropriate translation software. This paper describes the first operational CAPSL translator to the language used by the NRL Protocol Analyzer (NPA), a software tool developed specifically for the analysis of cryptographic protocols.

1 Introduction

The past few years have seen an increasing interest in the application of formal methods to the analysis of cryptographic protocols. Here are the reasons for this interest, and the response of the formal methods community.

1.1 Background

It is well known that protocols for exchanging cryptographic keys over data networks can be vulnerable to a class of attacks in which an intruder can intercept and alter messages in transit. In many cases, the security objective of a protocol can be subverted without "cracking" the cryptosystem, and the vulnerability is referred to as a *protocol failure.*

The abundance of failures discovered in published protocols led to the development of formal techniques for their security analysis. These techniques include the use of goal-directed state search tools implemented in Prolog, the application of general purpose specification and verification tools, and a specially-designed logic of belief, An outline of the history of the subject and some of the tools used can be found in [9]. Recently, search tools designed for hardware model-checking have been successfully applied in this area as well, such as [16] and related work mentioned in that paper. Recent work in inductive techniques was stimulated largely by Paulson [20]. A current survey of the area may be found in [8].

It became evident that it was difficult for analysts other than the developers of the various techniques to apply them. One reason for this difficulty is that the protocols had to be re-specified formally for each technique, and it was not easy to transform the published description of the protocol into the required formal system. Some tool developers began work on translators or compilers that would perform the transformation automatically. The input to any such translator still requires a formally-defined language, but it can be made similar to the message-oriented protocol descriptions that are typically published in articles, books, and protocol standards documents. A translator can also supply certain "boilerplate" material that changes little, if at all, from one analysis to the next, such as the specification of intruder capabilities.

This kind of thinking led to several languages: an early version of CAPSL [15]; ISL, from which CAPSL borrowed much of its style, supporting an application of HOL to an extension of the GNY logic [2]; CASPER [10], for the application of a model-checker FDR based on CSP as an input language; and Carlsen's "Standard Notation," from which per-process CKT5 specifications were generated [6]. The idea of making CAPSL into a common 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.

1.2 Implementation Plan

At this stage in the implementation of CAPSL, the language has a draft specification in the form of a technical report and a more recent description at a project web site [5]. The long-range plan for CAPSL includes developing an intermediate language (CIL, for CAPSL intermediate language), so that a single compiler from CAPSL to CIL can be shared by all users. Each tool will then need a translator from CIL to its own input language.

At present, CIL and associated tools are still under development. However, it was felt to be important to test the idea that an independently designed specification language could be used to support a pre-existing security analysis tool. This could be done by implementing an interim translator directly from CAPSL to the language of such a tool.

The first experiment along these lines is reported here: to develop an interface for CAPSL to the Protocol Analyzer developed at the U.S. Naval Research Laboratory (NRL).

1.3 The NRL Protocol Analyzer

The NRL Protocol Analyzer (NPA) is an interactive program, written in Prolog, that can be used to assist either in the verification of security properties or in the detection of security flaws. It has been used successfully on a number of protocols and has found unexpected flaws in several. It is currently being used to analyze the Internet Key Exchange Protocol [12, 13] and the Secure Electronics Transactions Protocol [14]. A more detailed description of a slightly earlier version of the Analyzer than the one discussed in this paper, that uses a somewhat different specification language, may be found in [11]. The current specification language is somewhat more high-level, but is still semantically consistent with the earlier one.

The analyzer requires protocols to be specified as a collection of rules expressing protocol state changes and symbolic reduction axioms characterizing encryption operators. These rules have a conditional statment form and use Prolog syntax for individual terms.

In order to provide a CAPSL interface for the analyzer, it was necessary to write a translator from CAPSL to the analyzer rule language. The translation process is divided into stages: (1) parsing the CAPSL into an abstract syntax tree and symbol table; (2) generating an internal semantic representation, including programs for each party in the protocol; and (3) generating the analyzer rules. The translator also generates other output representations.

The CAPSL translator for the NPA foreshadows the future universal CAPSL compiler in part, since the parsing stage and the internal semantic representation stage have essentially the same structure as the transportable compiler, and only the last stage is NPA-specific.

1.4 Summary of This Paper

In this paper, we give a brief taste of the CAPSL language, followed by a description of the analyzer and its specification language, with some observations on how the latter differs from CAPSL. We then describe the design and implementation of the translator, and finally present some conclusions and lessons learned.

2 The CAPSL Language

A CAPSL specification has three kinds of specification units: protocols, types, and environments. A protocol specification describes a protocol or subprotocol. A type specification introduces new encryption operators, if necessary. (Type specifications for standard encryption operators are provided automatically.) An environment specification provides scenario-specific details to set up an analysis session for model checkers or other tools that need them.

A protocol specification has three principal parts: declarations, messages and goals. Rather than attempt to cover the language in full here, we will just convey the flavor of CAPSL with an example.

The Otway-Rees protocol involves two communicating parties (initiator and responder) and a key server. The initiator sends a request to the responder, who forwards it to the key server. The key server sends two copies of the key (one encrypted with the initiator's master key, and one encrypted with the responder's master key) to the responder. The responder forwards the initiator's copy to the initiator.

The Otway-Rees protocol is fairly well known [19]; it has some subtle flaws that do not concern us here. We give the informal specification of the Otway-Rees Protocol first below. It has four messages.

1. $A \rightarrow B : M, A, B, \{N_A, M, A, B\}_{K_{AS}}$

   A initiates the protocol with B. It sends a session identifier M along with a certificate encrypted with a master key it shares with the server S. The certificate contains a nonce, the session identifier, and the names of A and B. In this protocol, a *nonce* is a key-sized randomly-generated value.

2. $B \rightarrow S : M, A, B, \{N_A, M, A, B\}_{K_{AS}}, \{N_B, M, A, B\}_{K_{BS}}$

B forwards the certificate to S, but also concocts a similar certificate of its own, in which it includes its own nonce. It encrypts this certificate with the master key K_{BS} that it shares with S.

3. $S \rightarrow B \ : \ M, \{N_A, K_{AB}\}_{K_{AS}}, \{N_B, K_{AB}\}_{K_{BS}}$

   S decrypts the certificates and checks that they have the same identifier. It then generates a session key. It encrypts the key together with A's nonce N_A with A's key, and the key together with B's nonce with B's key. Both results are sent to B together with M.

4. $B \rightarrow A : M, \{N_A, K_{AB}\}_{K_{AS}}$

   B decrypts its certificate and checks for its nonce. It forwards the remaining parts of the message to A. A decrypts its certificate and checks for its nonce.

We next give the CAPSL specification. As one can see, this is very close to the informal specification, the main difference being that the information given between the lines in the informal text is supplied formally in the declaration part of the specification. CAPSL also requires a formal statement of the security goals. Functional notation is used here for concatenation (`con`) and encryption (`se`), although CAPSL allows the use of the bracket notation as an option.

```
PROTOCOL Otway_Rees;
VARIABLES
    A, B, S: Principal;
    M: Field, FRESH;
    Na, Nb: Field, FRESH, CRYPTO;
    Kass, Kbs: Skey, CRYPTO;
    Kab: Skey, CRYPTO, FRESH;
    KeyTable(Principal): Skey;
DENOTES
    Kass = KeyTable(A);
    Kbs = KeyTable(B);
ASSUMPTIONS
    HOLDS A: A, B, S, M, Na, Kass;
    HOLDS B: B, S, Nb, Kbs;
    HOLDS S: S, KeyTable, Kab;
MESSAGES
1.  A -> B: M, A, B,
      se(Kass,con(Na, M, A, B));
2.  B -> S: M, A, B,
       se(Kass,con(Na, M, A, B)),
       se(Kbs,con(Nb, M, A, B));
    Kass = KeyTable(A);
    Kbs = KeyTable(B);
3.  S -> B: M,
       se(Kass,con(Na, Kab)),
       se(Kbs,con(Nb, Kab));
4.  B -> A: M, se(Kass,con(Na, Kab));
GOALS
    SECRET Kass: A, S;
    SECRET Kbs: B, S;
    SECRET Kab: A, B, S;
END;
```

Some of the declarations are just type declarations, and others reflect concepts peculiar to authentication protocol design. The nonces, for example, are declared with qualifications of `FRESH` and `CRYPTO`, meaning, respectively, that they have not been used before and that they are unguessable. These two properties are separable, since some protocols use nonces that are fresh but not unguessable (like TCP sequence numbers), and long-term keys are unguessable but not fresh.

The variable `KeyTable` and the `DENOTES` equations indicate how each party looks up the public key of other parties. The `HOLDS` assumptions show which variables are known initially by each party. The security objectives stated here are that the keys are shared secrets between A and B. One could also state other objectives, such as a belief by B that A holds K_{AB} at the conclusion of the protocol run, which would be expressed as `BELIEVES B: HOLDS A: KAB`.

The use of `Kass` instead of `Kas` is due to a naming conflict with a built-in use of `kas` as a key agreement operator. This problem is avoidable and will be fixed.

Note that in the `MESSAGES` section it is possible to interleave equations indicating tests or assignments between messages. It is also possible to put in assertions indicating message idealizations or intermediate goals. Furthermore, one may write conditional statements that invoke named subprotocols, using `IF-THEN-ELSE` and `INCLUDE` constructs.

3 The NRL Protocol Analyzer

The NRL Protocol Analyzer (NPA) is a tool for proving security properties of protocols and for finding attacks on protocols if the security properties do not hold. It works by having the user specify an insecure state: the translator works backwards from that state in an attempt to find a path starting in an initial state. Originally, the search space is infinite: the user reduces the search space to a finite one by proving a set of lemmas. The formulation and proof of these lemmas is not completely automatic, but a large amount of automatic assistance is provided, and we expect to provide more.

The model behind NPA is an extension of the Dolev-Yao model [7]. We assume that the participants in the protocol are communicating in a network under the control of a hostile intruder. The intruder has the ability to read all message traffic, destroy and alter messages, and create its own messages. Since all messages pass through the intruder's domain, any message that an honest participant sees can be assumed to originate from the intruder. Thus a protocol rule describes, not how one participant sends a message in response to another, but how the intruder manipulates the system to produce messages by inserting other messages.

Principals participate in a protocol by exchanging words. Words are the analyzer's alebraic term representations of message fields. They are created out of variables (denoted by strings beginning with a capital letter), atoms (denoted by strings beginning with a lower-case letter), and function symbols that take words as arguments. The result of applying a function symbol to a word or set of words is represented by the function symbol together with those words as arguments. Thus, if e denotes the encryption operator, the word $e(K, X)$ denotes the result of encrypting message X with key K.

We can now describe what an NPA specification looks like. A specification consists of several parts, the most important of which is a list of rules (called *protocol rules*) describing how participants send, receive, and process messages. A protocol rule involves four things: *words* that are input or learned by an intruder, *local state variables*, *local counters* that are incremented each time a protocol rule is fired, and *event statements* that are used to describe the action that occurred when a protocol rule was fired.

Each protocol participant possesses a counter representing its local time. It is initially set to zero, and is incremented by one every time a protocol rule in which it is involved is fired.

If A is an honest participant in a protocol, a *run of the protocol local to A* is defined to be a sequence of state changes local to A defined by the protocol designer. Each such state change defines a set of words produced by A and a set of changes to A's internal state variables. A run local to A is identified by the value of A's counter at the time the run begins. Any state variable local to that run is identified by the run's identifier. We note that a participant may participate in two or more runs concurrently. Since they will be identified by different local run values, they can be kept distinct.

All values of state variables are initially empty. The value of a state variable is computed as follows. Suppose that at time T, as recorded in the participant's counter, the value of a state variable S is Y. If a rule fires at time T making the value of S equal to Z, then the value of S at time $T+1$ is Z. If no such rule fires, then the value of S at time $T+1$ remains Y.

Besides describing the words and state variable values produced, each protocol rule also includes an optional *event statement*. The event statement gives a description of the event that occurred when that rule fired. Each event statement has five arguments. The first is the name of the principal to which the rule applies (this user can be the intruder), the second is a (possibly empty) list of other principals who may be involved in the event, the third is the name of the rule that produced the event, the fourth is a list of words involved in the event, and the fifth is the round number. Thus, for example, if we have a rule describing a server sending a key K to a principal U who has requested it, we might attach an event statement such as `event(server, [U], server_sendkey, [K], N)`. Event statements are used in defining queries that the user will present to NPA.

Note that it is up to the writer of the specification to decide what words are to be included in the event statement. There is no fixed algorithm for doing this; instead, it depends on what types of queries the user thinks he or she will be likely to make.

To give an example of an NPA protocol rule, consider the following hand-generated rule, which describes the first step in the Otway-Rees protocol:

```
Subroutine
  init_start(prin(A,honest),N,T):
init_self := prin(A,honest),
init_who :=  prin(B,H),
init_server := server(S,honest),
init_sessid :=
  rand(ts(prin(A,honest),N,T),sessid),
init_nonce :=
  rand(ts(prin(A,honest),N,T),nonce),
init_masterkey :=
  keytable(prin(A,honest)):
send msg ({init_self},{init_who},
  [{init_sessid},{init_self},
      {init_who},
     e({init_masterkey},
       ({init_nonce},{init_sessid},
        init_self},{init_who}))],N):
event(prin(A,honest),[{init_who}],
```

```
  init_start,
     [{init_nonce},{init_sessid}],N).
```

NPA protocol rules can also call other rules as subroutines. This is done by listing the names of the subroutines called in the order which they are called. Thus, for example, the sequence of actions performed by the server in the Otway-Rees protocol is specified by a rule that calls three actions as subroutines, as follows:

```
Session init_main(prin(A,honest),N,N):
init_start,
init_accept,
init_compromisekey.
```

Besides rules describing protocol transitions, an NPA protocol specification also contains sections describing which words are known initially by the intruder, what rewrite rules hold, what operations (such as encryption or decryption) are performed on words, and which of these operations may be performed by an intruder. These last are used by NPA to build a state machine model of the intruder.

The user is given a fair amount of freedom in defining words. However, there is one construct that MUST be used when words such as keys, nonces, etc., which must be random or pseudo-random, are defined. This is the name-round-time-stamp: `ts(A,N,T)`, where A is the name of the originator of the word, N is the round during which it was generated, and T is the time at which it was generated. If a name-round-time-stamp appears in a word, it is guaranteed to be different from any word generated by any other party, or during any other round, or at any other time.

Another construct that is not required, but is recommended, is the use of the honesty variable. A participant in a protocol can have an honesty variable attached to its name that can be set either to honest or dishonest. For example, we can identify a principal by `principal(A,H)`, where A is the principal ID, and H is set equal to "honest" or "dishonest". Honest principals, identified as `principal(A, honest)`, are assumed to follow the rules of the protocol and not to reveal their secrets unless the protocol requires it. Dishonest principals, identified as `principal(A,dishonest)`, are assumed to be in league with the intruder and to share all information with it. Thus, for example, if master keys are identified as `mskey(U)`, where U is a principal name, the intruder may be assumed to know all keys of the form `mskey(principal(A, dishonest))`. Finally, if an honest principal receives a message from another principal, it will not know whether the other principal is honest or dishonest; when an honest principal receives a message from another principal B, that principal will be represented by the name `principal(B, H)`, where H is an uninstantiated variable.

There are some important differences between CAPSL and the NPA language. In CAPSL, actions of parties upon receiving messages are usually left implicit. In the NPA language, they must be made explicit. Moreover, in CAPSL, properties such as freshness, uniqueness, randomness, etc. are declared initially. In the NPA language, they must be guaranteed by the use of name-round-time stamps. In CAPSL, intruder actions and knowledge are left unspecified. In the NPA language, they are defined, if only implicitly. Finally, CAPSL has nothing resembling the event statement used in the NPA language. Thus, a CAPSL-to-NPA translator has a large amount of information to generate.

4 The Translator

The CAPSL translator inputs CAPSL specifications and outputs specifications in different, user-selected, forms suitable for different types of use. This section describes the overall structure of the CAPSL translator and how it produces its outputs.

The current version of the translator produces outputs in the following three forms, of which only the first is used in the current effort:

- specifications for NPA;
- specifications for a preliminary version of the Protocol Description Logic (PDL), a Higher Order Logic (HOL) theory of authentication protocols and protocol failure; and
- human-readable descriptions of the algorithms followed by the processes implementing a protocol's roles.

See [3] for the preliminary version of PDL used by the translator and [4] for a later version of PDL intended to formalize new CAPSL capabilities.

The remainder of this section describes the CAPSL translator's structure and algorithms, particularly its algorithms for producing NPA specifications.

4.1 Translator Structure

The CAPSL translator is a `flex/bison` application; `flex` and `bison` are standard compiler-writing tools, available from the Free Software Foundation, for generating lexical analyzers and parsers, respectively. The translator operates in three stages:

- Stage 1: Parse the CAPSL specification, along with definitions of default CAPSL Abstract Data Types, and create corresponding internal data structures.
- Stage 2: Compute the algorithms, expressed in further internal data structures, carried out by processes performing each of the specified protocol's roles.
- Stage 3: Produce output of the user-selected type from the data structures computed in Stage 2.

Detailed descriptions of each of these stages follow.

4.2 Stage 1

Stage 1 first internally prepends the standard TYPESPEC specifications for built-in encryption operators to an input CAPSL specification file, then parses the augmented file. The translator produces internal data structures, most notably a symbol table, that describe the contents of the augmented file.

The entry for a symbol in the translator's symbol table completely defines the object that this symbol names. In particular, the entry for a protocol identifier contains lists of actions that describe all the actions taken in following the protocol, treating these actions as if they were always sequential.

In addition to producing the symbol table, Stage 1 identifies syntactic errors and some semantic errors in the CAPSL specification file, particularly type incompatibilities and symbols that are used before they are defined. Stage 1 also implements syntactic CAPSL capabilities that allow symbols to rename other symbols, abbreviate expressions, and represent subprotocols.

4.3 Stage 2

Stage 2 expresses the algorithms followed by the processes performing a specified protocol's roles as lists of statements in a simple, unnamed, C-like, imperative programming language. Stage 2 expresses these statement lists as internal data structures; Stage 3 produces text output of the user-selected form from these data structures. Informal descriptions of the different statement types in this unnamed internal language, and their meanings, follow.

In these descriptions, a *slot* is an abstract storage location capable of holding any finite amount of data of any type, corresponding roughly to a CAPSL variable. An *expression* is either a slot name or a term consisting of an algorithm name applied to a tuple of expressions. Distinct processes' slots are distinct, though different processes' slots can have the same names.

- ASSIGN <slot name> <expression>. The statement replaces the current contents of the slot with the value of the expression. When used on the right-hand side of an assignment statement, a slot name denotes the contents of this slot.
- IF <expression> THEN <statement list 1> ELSE <statement list 2>. If the expression is true, the statement does the first statement list, and otherwise it does the second statement list.
- RECEIVE. The process executing a RECEIVE hangs until a message addressed to this process arrives, then stores this message in a special slot named temp. The CAPSL translator assumes that every message's first two fields are the names of its purported intended recipient and its purported sender.
- SEND <expression 1> <expression 2>. The statement sends the process named by the first expression the message consisting of the receiving process' name, the sending process' name, and the value of the second expression.
- TEST <expression>. This statement evaluates the expression, and if it is true does nothing. Otherwise, it raises an alarm and aborts the protocol.

Although this unnamed intermediate language exists only internally to the CAPSL translator, the preliminary version of PDL gives a HOL formalization of it, and the translator's "human-readable description" output is essentially a textual form of it.

Stage 2 determines the initial contents of each processes' slots and the subsequent computations that each process performs. The essential problems it solves are interpreting CAPSL equalities and making CAPSL default actions explicit.

CAPSL equalities are problematic because they are sometimes assignments and sometimes equality tests; further, if they occur between concatenations, they can mean multiple assignments or equality tests, or mixtures of assignments and equality tests. CAPSL interprets an equality as a test if both the values set equal are defined, and interprets it as an assignment if only one of these values is defined.

Stage 2 enforces that all distinct computed expressions are stored in distinct slots, maintains lists of slots that definitely, possibly, and definitely do not have assigned values, and uses these lists to interpret CAPSL equalities. It takes intersections and unions of these lists to bound the effects of IF-THEN-ELSE statements. Stage 2 also uses these lists to determine if a slot's contents might be used before they are defined, and if so raises a CAPSL semantic exception. Stage 2 also raises a CAPSL semantic exception if an assignment might change a `CONSTANT` value, change a non-temporary slot value, or invalidate a relationship given by a `DENOTES` statement.

A process can perform CAPSL default actions whenever it receives a message or performs an assignment to a slot whose contents it had not yet determined. Stage 2 assumes that a process performs equality tests on any values it receives that the process already holds. Stage 2 further assumes that a process decrypts any encrypted values it holds that it is able to decrypt, applies the inverses of functions to any values it holds that are computed with these functions, and performs equality tests on any values it obtains that the process already holds.

4.4 Stage 3

This subsection describes the portion of the CAPSL translator's Stage 3 code that produces output for the NPA. The remainder of this subsection will use "Stage 3" to mean "the portion of the translator's Stage 3 code that produces NPA output". As the previous subsection indicated, the translator's human-readable and PDL outputs are straightforward variants of the translator's internal imperative programming language, so producing them is much simpler. The essential problem solved by Stage 3 is translating algorithms in a C-like imperative programming language into the Prolog-based specification language of NPA.

Stage 3 first checks that all the protocol's algorithms are ones that the NPA can analyze. It raises a semantic error if an algorithm uses any associative operation or any commutative operation that is not specifically handled by NPA rules.

As a convenience to users, Stage 3 checks that each function symbol or operation used in a protocol is at least mentioned in an `AXIOM` or `DENOTES` statement for the protocol, and prints a warning that there will be no way for NPA to make inferences about the function symbol or operation if not.

After finding all the identifiers that must be declared in the NPA translation of a protocol's algorithms — a complicated process that involves partitioning the protocol's algorithms into fragments that are expressible as NPA subroutines — Stage 3 begins outputting the text of the NPA translation. It declares the following identifiers specific to the protocol, along with other, protocol-independent, declarations needed by NPA:

- event names naming the translation's `Session` and `Subroutine` elements;
- state variable names giving the protocol's slots;
- variables used to represent the possible honesty values of the processes performing the protocol's roles;
- two time variables for each processe playing one of the protocol's roles — the NPA uses one time to identify sessions, and another time to identify sequential events within a session;
- Prolog variables used to temporarily store elements of received or assigned lists before these elements are stored in state variables;
- function and operation symbols specific to the protocol;
- `RandID` atoms specific to the protocol.

Stage 3 then outputs NPA rewrite and commutativity rules, rules that essentially just reformulate definitions in the CAPSL default type specifications giving properties of standard functions used in the protocol.

Stage 3 next outputs a list of values assumed to be known to an attacker trying to defeat the protocol. It generates this list by examining each term denoting a value used in the protocol and computing the most general form of this term that will be known to an attacker, assuming that the attacker knows all names, all times, all public keys, and all information known to any dishonest protocol process.

Finally, Stage 3 outputs the `Session` and `Subroutine` definitions for each of the protocol's algorithms. It defines each `Session` to contain assignment statements that compute the values in the algorithm's initialized slots, followed by calls to NPA subroutines that carry out the various NPA-expressible parts of the algorithm. It expresses each if-then-else statement as an NPA subroutine call of the form `S1: Or: S2` where `S1` is a subroutine that executes only if the condition in the if-then-else statement is true and `S2` is a subroutine that executes only if the condition in the if-then-else statement is false.

5 Results of Using the CAPSL-to-NPA Translator

We tried out the translator on a number of different protocols, but wound up concentrating on six: Kerberos[18], the two Needham-Schroeder protocols [17], Otway-Rees[19], Bellovin and Merritt's Encrypted Key Exchange (EKE)[1], and a fragment of the Secure Socket Layer Protocol[21]. We had already developed hand-generated NPA specifications of Needham-Schroeder public key and the Otway-Rees protocol (two rules of which are given in Section 4), so we found these useful for doing comparisons between translator and hand-generated output.

Somewhat to our surprise, we found the translator output very similar in appearance to that generated by humans. The main difference was that state variable names, which were generated automatically, were not that helpful in identifying to the reader that part of the protocol to which they applied. Also, the event statements, which contained all state variables used in a round, were considerably more bulky. But other than that, there was not much difference between the look of a translator- and a human-generated specification.

For example, consider the following two rules in the translation of the CAPSL specification of the Otway-Rees protocol. The first rule, `a_main`, describes the order in which the initiator's rules are executed, and also defines the terms that are generated by the initiator. The second, `a_sub_1` describes the initiator sending the first message. Thus `a_main` and `a_sub_1` correspond, respectively, to the hand-generated rules `init_main` and `init_start` in Section 3.

```
Session a_main(prin(Ua,honest),Ta1,Ta1):
Session a_main(prin(Ua,honest),Ta1,Ta1):
a_A := prin(Ua,honest),
a_B := prin(Ub,Hb),
a_KeA := keytable(prin(Ua,honest)),
a_M := freshfield(ts(prin(Ua,honest),
  Ta1,Ta1),am),
a_Na := newcryptofield(ts(prin(Ua,honest),
  Ta1,Ta1),ana):
a_sub_1,
a_sub_2.

Subroutine a_sub_1(prin(Ua,honest),Ta1,Ta2):
a_Kass := {a_KeA}:
send msg({a_A},{a_B},
  [{a_M},{a_A},{a_B},
  se({a_Kass},
    ({a_Na},{a_M},{a_A},{a_B}))],Ta1):
event(prin(Ua,honest),[{a_B}],a_sub_1,
  [{a_A},{a_B},{a_Kab},{a_KeA},
  {a_M},{a_Na},{a_seKacoNaKa}],Ta1).
```

Efficiency was another concern. There were two main ways that a translator-generated protocol differed from a human-generated one. One was that the translator often broke a simple state transition into several subtransitions. The other was that words used in a protocol were always created at the beginning of a protocol run by the translator, while human specification writer could specify parties as creating words any time before they were used. Thus, for example, in the above specification, the nonce denoted by `freshfield(ts(prin(Ua,honest), Ta1,Ta1),am)` is generated in `a_main`, while in the hand-generated rules given in Section 3 the same nonce, denoted by `rand(ts(prin(A,honest),N,T),nonce)` is generated in the second rule, `init_start`. This difference tends to make the hand-generated specification more efficient, for the following reason: the Analyzer includes a mechanism for discarding states that contain name-time-stamp words that are used be before they were created. Since the Analyzer works by searching backwards, these types of unreachable states are detected earlier when the the name-time-stamp words are created later in a specification.

We expected both of the above described differences to affect efficiency. Actually, only the second seems to. When we used the NPA to process queries on the translated Needham-Schroeder protocol, we found very little difference in efficiency between it and the human-generated one. However, when we ran the translated Otway-Rees protocol, we found it considerably slower than the human-generated one. When we altered the specification so that words involving name-round-time stamps were generated immediately before use, though, this difference in efficiency went away, and the translated specification became as efficient as the hand-generated one. It is not quite as straightforward for the translator to generate words right before use as it is to put them at the beginning of the protocol, but, given the dramatic improvement in efficency it brings, we think that this change is well worth implementing.

Finally, we were concerned about expressiveness. In this, we had mixed results. However, the problems we have encountered have shown us ways in which both the translator and CAPSL can be improved. The six specifications we used introduced various features. Kerberos is based on timestamps. The Otway-Rees, the Needham-Schroeder protocols, and Kerberos make use of key servers. EKE uses the Diffie-Hellman protocol, a protocol with commutativity properties that allowed us to try out an experimental implementation of commutativity for NPA. SSL is a complex protocol with several options; we used it to test our use of `IF-THEN-ELSE`.

The first problem we encountered was that the translator did not allow for comparisons between variables and constants, which are used for IF-conditions. A mechanism to do this is being implemented.

The next problem we encountered was that we could not specify any assumptions about honesty. In most of our hand-generated specification of server-assisted protocols, we have assumed that the key server is always honest; clearly, if the key server is not honest the protocol will fail. CAPSL allowed us to make no such distinction, and so the translator generated both honest and dishonest key servers. However, this does not appear to cause a serious problem in either efficiency or expressiveness: the user can simply specify that actions must involve an honest key server when presenting his or her query.

Another problem we found was in the expression of key compromise. At the beginning, we had no standard way of representing key compromise in Analyzer specifications, and no way of recognizing what words should be vulnerable to compromise. Since then, we have worked out a standard representation of compromise and a standard way of recognizing vulnerable words in a CAPSL specification. The vulnerable words are those declared to be `SECRET` in the goals section (so we know which parties are supposed to possess them as secrets), and `FRESH` (so they are relevant only to a single session). It should then be simple to generate compromise transitions for the parties who are intended to possess the secrets. We do not anticipate any problem in having the translator recognize these words and append compromise transitions to the end of a specification.

6 Conclusions and Lessons

We made a number of changes to CAPSL as a result of this experience of building an interface to NPA. Some changes were made immediately to help us through the translator project, and others had their impact in the form of more radical language revisions that are still in progress. The deeper changes concern the design and use of the specification units we have not discussed here, type specifications and environments.

The immediate changes included the introduction of the `LONGTERM` property for a variable, meaning that it has the same value in several sessions. Also, we added a security goal `SESSION_SECRET`, which differs from `SECRET` in that the secret item is no longer considered sensitive after a session has terminated.

Many needs for specifying a protocol's environment, including different styles for giving principals' roles, were worked out during design discussions for future versions of CAPSL. We found, for example, that sometimes it is desirable to consider "what-if" scenarios in which the intruder is assumed to possess secret keys or other specific items that would normally not be available. An `ATTACKER_HOLDS` section was added for this purpose.

Some aspects of the language were found to be awkward or inadequate, and while no changes in them were made immediately, these discoveries had an effect on the new design in progress. One problem is how to indicate whether functions declared in type specifications are available for use by the intruder: Encryption functions are usually assumed to be known to the intruder, but a function that produces the secret key of a principal is also declared in a type specification, and the intruder cannot evaluate it to obtain secret keys. A property `PRIVATE` for these functions was planned.

The choice of keywords and other details of the syntax can probably be improved. It has been pointed out that certain keywords are overloaded. The true meaning of `VARIABLES` and `CONSTANTS` in a protocol definition, for example, is different enough from both programming language concepts and abstract datatype concepts that other keywords might be more appropriate; and the `DENOTES` section is used for two or three conceptually different purposes.

The language is still evolving, with the help of contributions from the community of researchers. The current version, documented on the web site [5], reflects the needs and insights that were manifested during this effort.

References

[1] Steven Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attack. In *Proceedings of the IEEE Computer Society Symposium on Research on Security and Privacy*, pages 72–84. IEEE Computer Society Press, 1992.

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

[3] S. Brackin. A state-based HOL theory of protocol failure. Technical Report 98007, Arca Systems, Inc., Ithaca, NY, October 1997.

[4] S. Brackin. A HOL formalization of CAPSL semantics. In *Proceedings of the 21st National Conference on Information Systems Security*. IEEE, October 1998.

[5] CAPSL: Common Authentication Protocol Specification Language. http://www.csl.sri.com /~millen/capsl.

[6] U. Carlsen. Generating formal cryptographic protocol specifications. In *1994 Symposium on Research in Security and Privacy*, pages 137–146. IEEE Computer Society, 1994.

[7] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[8] S. Gritzalis, D. Spinellis, and P. Georgiadis. Security Protocols over open networks and distributed systems: Formal methods for their analysis, design and verification, Computer Communications, 1999, to appear. Currently available at http://kerkis.math.aegean.gr/~dspin/pubs/jrnl/1997-CompComm-Formal/html/formal.htm.

[9] Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 3(2), 1994.

[10] G. Lowe. Casper: a compiler for the analysis of security protocols. In *10th IEEE Computer Security Foundations Workshop*, pages 18–30, IEEE Computer Society, June 1997.

[11] Catherine Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[12] Catherine Meadows. Using the NRL Protocol Analyzer to examine protocol suites. In *Electronic Proceedings of LICS Workshop on Formal Methods and Security Protocols*. http://www.cs.bell-labs.com/who/nch/fmsp/program.html, 1998.

[13] Catherine Meadows Analysis of the Internet Key Exchange Protocol Using the NRL Protocol Analyzer In *Proceedings of the 1999 Symposium on Security and Privacy*, to appear.

[14] Catherine Meadows and Paul Syverson. A formal specification of requirements for payment transactions in the SET protocol. In *Proceedings of Financial Cryptography '98*, pages 122-140. Springer-Verlag Lecture Notes in Computer Science, 1998

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

[16] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols. In *1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997.

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

[18] B. Clifford Newman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[19] David Otway and Owen Rees. Efficient and timely authentication. *Operating Systems Review*, 21(1):8–10, 1987.

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

[21] The SSL protocol. http://home.netscape.com /newsref/std/SSL.html, March 1996.