Paper

CAPSL and MuCAPSL

Jonathan K. Millen and Grit Denker

Abstract — Secure communication generally begins with a connection establishment phase in which messages are exchanged by client and server protocol software to generate, share, and use secret data or keys. This message exchange is referred to as an authentication or key distribution cryptographic protocol. CAPSL is a formal language for specifying cryptographic protocols. It is also useful for addressing the correctness of the protocols on an abstract level, rather than the strength of the underlying cryptographic algorithms. We outline the design principles of CAPSL and its integrated specification and analysis environment. Protocols for secure group management are essential in applications that are concerned with confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership and access control. We will also discuss our progress on designing a new extension of CAPSL for multicast protocols, called MuCAPSL.

Keywords — CAPSL, MuCAPSL, cryptographic protocol specification, cryptographic protocol analysis, secure group communication, multicast.

1. Introduction

In computer networks, cryptography is used to protect private messages and to authenticate the source and content of messages. The range of applications of cryptographic techniques is enormous, including banking, electronic commerce, protection of personal and medical data, trade secrets, and government and military uses. Cryptography supports secure access to World-Wide-Web servers, virtual private networks, and other services over the Internet.

Secure communication generally begins with a connection establishment phase in which messages are exchanged by client and server protocol software to generate, share, and use secret data or keys. This message exchange is referred to as an authentication or key distribution protocol. A few such protocols have been put forward by standards bodies, and others are in common use, such as SSL for securing web page accesses, but new protocols are continually being designed. The perpetual need to design new protocols is due to new technology – cryptographic algorithms, computer hardware, network architectures – and new applications, with special goals such as digital cash, voting or contract signing.

The principal topic of this paper is the design of a formal language, CAPSL, for specifying cryptographic protocols, and how this language plays a role in the analysis of their correctness. CAPSL is useful for addressing the correctness of the protocols on an abstract level, rather than the strength of the underlying cryptographic algorithms. We will also discuss our progress on designing a new extension of CAPSL for multicast protocols, called MuCAPSL.

1.1. Protocol vulnerabilities

Protocols can be analyzed under the assumption of ideal encryption: that is, ciphertext can be decrypted only with the help of the proper key, and ciphertext for a chosen plaintext cannot be generated without the help of the proper key. These assumptions distinguish formal models, with which we are concerned, from computational models, which apply probabilistic and computational complexity reasoning, and from cryptanalysis. We assume that the reader is aware of the distinction between public-key encryption, in which an encrypting key is publicized and the corresponding decryption key is kept private, and symmetric-key encryption, for which the two parties share a common secret key.

Cryptographic protocols are designed to defend against hackers or other adversaries who may have the ability to intercept and modify messages on the network. The attacker may also have a legitimate user identity on the network, or (almost the same thing) may have compromised the secret key of some legitimate user in order to masquerade as that user. Protocol designers also consider attacks in which some secret keys that have been in long-term use, or which were used in the past, may have become compromised due to cryptanalysis, and the protocol should be designed so that such compromised keys cannot be re-introduced or used to compromise new keys. This concept of a worst-case, powerful attacker originated in a paper by Dolev and Yao [15]. Attacks perpetrated by a Dolev-Yao attacker are called *active, message-modification,* or sometimes *man-in-the-middle* attacks.

Here is a well-known example of an authentication protocol, showing how such protocols are expressed in textbooks and papers. We will also show how this protocol is vulnerable to a Dolev-Yao attacker:

$$A \to B : \{A, N_a\}_{PB}$$
$$B \to A : \{N_a, N_b\}_{PA}$$
$$A \to B : \{N_b\}_{PB}$$

This particular protocol is supposed to establish a session between *principals* A and B in such a way that each principal authenticates the identity of the other principal, and they share two session-specific secrets N_a and N_b. This protocol was proposed by Needham and Schroeder in [27]. What is shown here is actually not the entire protocol, but just the handshake that comes after an earlier part in which the necessary public keys are exchanged. We will refer to this protocol as "NSPK."

The bracketed term $\{A, N_a\}_{PB}$ represents the encryption of the concatenation of A and N_a using the public key of B. It is assumed here that A has previously obtained B's public key and that only B has the corresponding secret key, and vice versa for B. The message fields N_a and N_b are *nonces,*

meaning that they are *fresh*, in the sense that they have not been used before by the principal that originates them. If they are large enough and randomly generated, they could be used as keys to encrypt subsequent messages.

The security claim of this protocol is that A has given N_a directly only to B, because only B could have decrypted the message in which N_a was introduced. Similarly for B and N_b. The protocol also provides entity authentication, i.e., evidence that the other principal is currently actively participating in the protocol, because it includes acknowledgments from B and A containing the nonces they received.

This abstract message-list style of protocol presentation is often called an "Alice-and-Bob" specification, from the conventional names given to the parties represented by A and B.

There is an active attack on the Needham-Schroeder protocol, found by Lowe [19]. Lowe's attack is illustrated in Fig. 1.

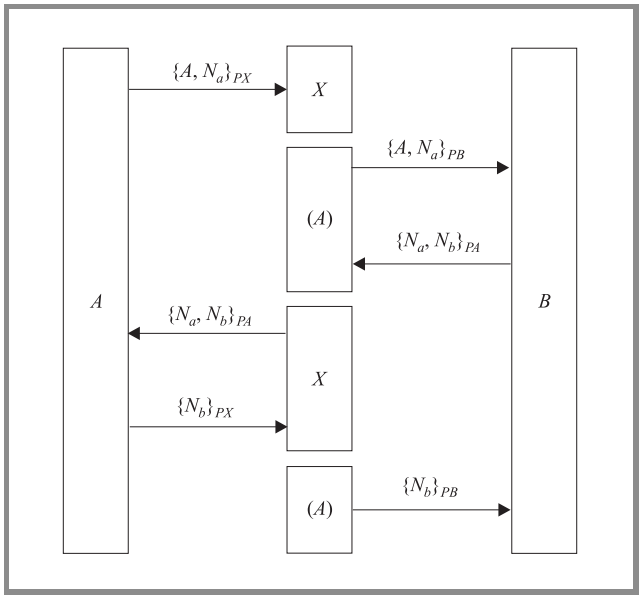

Fig. 1. Lowe's attack.

In this figure, the center column represents the intruder playing two roles. One role is as himself, principal X, responding to A in the left-hand session of the protocol. The intruder is also *masquerading* as A in the right-hand session of the protocol, indicated with (A) in parentheses. There is a security breach in the right-hand session, because B ends up believing he has been talking to A, and that N_b is shared only with A.

1.2. Formal methods

The existence of active attacks led to the development of methods to detect them. Several approaches have been developed, such as specially designed goal-directed state search tools [21, 22] to find attacks, applications of general-purpose specification and verification tools [5, 18, 30] to perform inductive proofs of correctness, specially designed logics of belief [2, 16] to prove authentication properties, and applications of model-checking tools [9, 19, 33], also to search for attacks. These are some of the earliest or most influential papers, and by now the literature is quite extensive.

These tools and their successors have been effective, but it is difficult for analysts other than their developers to apply them. One reason for this difficulty is that the protocols must be respecified for each technique, and it is 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 Alice-and-Bob specifications. This is the CAPSL approach. The origins of CAPSL were at the 1996 Isaac Newton Institute Programme on Computer Security, Cryptology, and Coding Theory at Cambridge University.

This approach was also taken by ISL, supporting an application of HOL to an extension of the GNY logic [6]; Casper [20], for the application of FDR using a CSP model-checking approach; and Carlsen's "Standard Notation" [7], which was translated to per-process CKT5 specifications [4].

1.3. The CAPSL approach

The CAPSL language and supporting tools are still under development. This document discusses the design concepts of the language, including the strategy by which CAPSL can be adapted for use by various protocol analysis tools. The basis of this strategy is the use of an intermediate language, CIL, that is close to the state-transition representation used by almost all of these tools. 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 a variety of tools.

CAPSL is parsed and translated to CIL, and there are different translators, called *connectors,* from CIL to whatever form is required for each tool. The translator from CAPSL to CIL can deal with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the underlying separate processes. Connectors deal with the semantics and requirements of individual tools. This overall plan is summarized in diagram shown on Fig. 2.

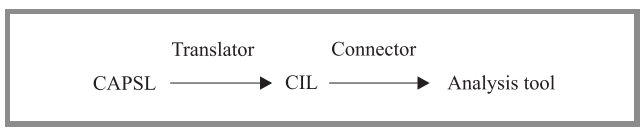

Fig. 2. CAPSL translation.

An overview of the CAPSL and CIL environment was given in [11]. The reference report specifying CAPSL is [13], and there is also a web site with CAPSL information [25].

2. CAPSL overview

The acronym "CAPSL" stands for "Common Authentication Protocol Specification Language." A CAPSL specification is made up of three kinds of modules: *TYPESPEC*, *PROTOCOL*, and *ENVIRONMENT* specifications, usually in that order. Typespecs declare cryptographic operators, hash functions, and other operations axiomatically as abstract data types. Specifications for the most popular operators, representing the abstract features of cryptosystems like DES, RSA, and Diffie-Hellman, are included in a standard "prelude" file of typespecs supplied with the CAPSL environment.

Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools. We will not discuss environment specifications here.

The core of a protocol specification is a message section containing an Alice-and-Bob specification of the protocol.

An important part of the protocol specification is a statement of its security objectives. There is a *GOALS* section for this purpose, which may include secrecy and authentication statements. Initial assumptions are also specified formally and placed in a section prior to the message list.

Here is a protocol specification for NSPK:

```
PROTOCOL NSPK;
VARIABLES
  A,B: PKUser;
  Na,Nb: Nonce, CRYPTO;
ASSUMPTIONS
  HOLDS A: B;
MESSAGES
  1. A -> B: {A,Na}pk(B);
  2. B -> A: {Na,Nb}pk(A);
  3. A -> B: {Nb}pk(B);
GOALS
  SECRET Na
  SECRET Nb;
  PRECEDES A: B | Na;
  PRECEDES B: A | Nb;
END;
```

Note that declarations may contain property keywords, such as CRYPTO, having some semantic significance. CRYPTO for the variables N_a and N_b means that their values are not guessable (by an attacker). This is significant during analysis, for the attacker model. A variable of type Nonce is assumed by default to have the property FRESH, meaning that values chosen for it have not been used before by the same principal. A nonce is not necessarily CRYPTO, since sequence numbers, i.e., numbers that are increased by one, are FRESH and guessable.

The HOLDS declaration states that the process executing on behalf of *A* has been initialized with the principal *B* chosen as the responder. If the HOLDS assumption is omitted, the CAPSL translator will complain that the sender of the first message does not know the receiver address. By convention, principals always hold themselves.

2.1. Key lookup

Note that the public keys *PA* and *PB* in NSPK have been replaced by function calls `pk(A)` and `pk(B)`. While *A* could have been initialized with *PB*, *B* needs a table lookup, represented by `pk(A)`, to find *PA*, since *B* does not know in advance who will request a connection.

Declaration of key lookup functions is one of the main uses of the abstract type specifications in CAPSL. Such functions are defined for different subtypes of Principal. They embody the kind of long-term key memory a certain kind of principal is assumed to have.

The function `pk` is defined on principals of type PKUser, assumed to have such a table. Principals of type PKUser also have a function `sk` to look up their own corresponding private key. Thus, there is a typespec as follows:

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

The typespec name, in this case PPK, is distinct from the name of the type or types declared in it. Typespec names are used in IMPORTS statements, when necessary. Presently the CAPSL translator does not import typespecs by name; it simply accepts whatever typespecs are provided in its input stream, in order, and requires that a symbol be declared before it is used. Declarations are global, including those of dummy variables used in axioms.

Functions in type specifications are public by default, meaning that both honest principals and the attacker may compute them. However, some functions, like `sk`, deliver long-term secrets such as private keys. These are declared using the keyword "PRIVATE". The first argument of such functions identifies the principal privileged to hold the function value.

There are two kinds of axioms: equational axioms and INVERT axioms. Equational axioms specify the declared functions for the use of theorem provers or term rewriting systems, and also for the use of human readers of the specification, to determine whether the declared abstractions are

suitable as a model for the "real" functions in the protocol. The INVERT axioms are used by the CAPSL translator to determine implementability of a protocol, which will be discussed later.
There are other typespecs in the prelude for principals holding long-term shared symmetric keys, and still others can be added by CAPSL users as needed.

2.2. Goals

Presently two kinds of security goals are supported: SECRET and PRECEDES. A goal `SECRET K;` means that protocol variable K should not be obtainable by the attacker (unless the attacker is acting overtly as one of the legitimate principals in a particular protocol session).
A goal `PRECEDES A: B | K, N;` means that when a principal B reaches the final state of a protocol session, there must be some session of principal A (not necessarily in its final state) that agrees with B on the values of A, B, K and N (or any other variables listed after the |, if any). This security goal is meant to represent a fairly general kind of authentication, and it corresponds roughly to formalizations of authentication used by Schneider [34] and Lowe [20].

2.3. Concatenation

A sequence of fields may be concatenated into a single longer field, usually for the purpose of having them encrypted together. Curly brackets { , } and square brackets [,] denote different kinds of concatenation, which are represented by different functions, `cat` and `con` respectively. `cat` is associative and `con` is not.
Both `cat` and `con` are binary. Longer concatenations are parsed under the assumption that right association is intended. Thus, $[a,b,c]$ is parsed as $[a,[b,c]]$.
Associativity of concatenation matters when we try to decompose a concatenation. In particular, the first component of a `cat` term is extracted by `first`, and that of a `con` term by `head`. It is easy to characterize `head` with the axiom `head(con(X,Y)) = X`. But `first(cat(`X`,`Y`))` could be either X or `first(`X`)`, depending on whether or not X is itself a `cat` term.
To deal with this question we differentiate between *atomic* fields, which form the subtype Atom of Field, and those fields that are expressible as a concatenation of smaller fields. The first component of a `cat` concatenation is the first atomic component. Most types are subtypes of Atom.
Another feature of associative concatenation is that a message `A -> B: {C,D}K` can be received by B only if either (1) C is held by B or (2) C is atomic. If C is neither held or atomic, B cannot parse the concatenation from left to right – it won't know where C stops and D begins.

2.4. Other language features

CAPSL has additional syntax to make it more expressive, more concise, and to help resolve ambiguity.

Variables can be introduced to precompute expressions. Suppose, for example, a certain symmetric key K is computed as a hash of other variable values. We can write an equation that looks like an assignment statement:

```
K = sha({Na,A,Kab});
```

This equation can be placed in the MESSAGES section before a message containing the first use of K. Alternatively, it can be placed in a prior DENOTES section, like a declaration, and it will automatically be used when needed. (This is done with a preprocessing step in the CAPSL translator.) If K is computed in two different ways by different principals (this might happen, for example, when computing a Diffie-Hellman shared key), each DENOTES equation can be labelled by the principal allowed to use it, e.g.,

```
K = sha({Na,A,Kab}):A;
```

Another useful feature is the % syntax introduced by Lowe in Casper [20]. A message might be computed with an expression by the sender, but handled by the receiver as a black box. The sender of a term $X\%Y$ views it as X but the receiver sees it as Y. Consider the statements:

```
A -> B: {X}pk(C)%Y;
B -> C: Y%{X}pk(C);
```

While A and C understand the message as an encryption, B merely forwards it.
Message sections may also invoke a subprotocol (specified separately as a protocol) using an INCLUDE statement, and make tests using equations to either abort the protocol on a failed test or to choose between IF-THEN branches.
It is one of the characteristics of CAPSL as a specification language that protocol variables receive a value only once, and are not changed after they have been initialized or computed or received. This means that an equation like `K = sk(B);` can be unambiguously identified as either an assignment statement (if K is not defined but B is), a test (if K and B are both defined), or a mistake.

3. The intermediate language CIL

The CAPSL intermediate language (CIL) is designed to make the translation to tool-specific representations as easy as possible. Fortunately, the protocol specifications required for most protocol analysis tools have considerable structural similarity. They generally specify a protocol with state-transition rules for communicating processes. CIL uses multiset term rewriting rules that permit state changes to be presented concisely, and in a way that closely matches the requirements of analysis tools. This approach was influenced by an analysis example using Maude, by Denker, Meseguer, and Talcott, presented at a LICS '98 workshop [14], and by Mitchell's multiset rewriting (MSR) formulation, presented at a Computer Aided Verification workshop in 1998, and also later, in more detail, in [8].

3.1. The MSR model

In the MSR model, the current global state of a network is a multiset containing "facts" representing the current state of some processes engaging in the protocol, and some messages in transit. The network is a multiset simply because it is possible that many copies of the same process state or message might be present simultaneously due to multiple concurrent protocol sessions.
An MSR rule for a state transition in which A handles the exchange

```
B -> A: {K}pk(A);
A -> B: {A,Na}K;
```

might have the abstract form:

$$A_i(A,B), M(\{K\}_{\mathtt{pk}(A)}) \longrightarrow$$
$$(\exists N_a) A_{i+1}(A,B,K,N_a), M(\{A,Na\}_K).$$

Here, "A" is being used both to name a role in the protocol, with states A_i, and as a dummy variable in the rule. The arguments of a state fact are the variables (identified by their positions rather than their names) held by the process. The first argument is always the principal running the process. Message facts are of the form $M(\_)$. The parameter of the message fact holds the content of the message.
In this rule, A in state i decrypts the received message (on the left), adds K to its memory list for the next state $i+1$, generates a nonce N_a, and replies with $\{A\}_K$. The message facts in this rule show only the content of a message, not its source and destination "header." By convention, facts appearing on the left but not on the right are removed from the multiset when the rule is executed.
The existential quantifier in the MSR rule has more than its ordinary logical meaning: it asserts that any value chosen for the variable is a fresh value. This is like a skolemization step when a new constant is chosen to instantiate an existentially quantified variable for proof purposes.
Note that two CAPSL messages have been used to produce a single MSR rule. We could have written two MSR rules for A, one to receive the first message and one to send the second, but then the two rules can be combined. The CAPSL translator actually does this; it processes one message at a time, producing receive-only and send-only rules, and then combines compatible pairs in an optimization step [12].

3.2. CIL vs. MSR

CIL is a variant of MSR. CIL represents state facts in the form

```
state(roleA,i,terms(A,B))
```

and messages in the form

```
msg(B,A,ped(pk(A),K)).
```

Syntactically, these are simply function-term presentations of an abstract syntax tree. Lower-case symbols are node labels, usually function names, and upper-case symbols are variables. Note that the functional representation of `{K}pk(A)` is `ped(pk(A),K)`, and that the CIL version of a message *does* include the source and destination principals.
For the sake of readability, we will continue to use the MSR rather than the CIL form of rules for subsequent explanations.
Another difference between MSR and CIL is that the CIL output of the CAPSL translator includes additional information that is potentially useful for analysis tools, such as a symbol table containing the type signatures of all variable and function names.

3.3. Goals

Presently, goal declarations are translated more or less literally from the CAPSL form to the CIL form, for later use by tool connectors. It is possible to do more, because goals that are security invariants can be converted to MSR and CIL rules that recognize insecure states of the multiset and trigger a "violation" fact. This approach is not difficult to apply manually in a protocol-specific way, but it is is not so easy to set up a general goal translation approach that works for all protocols, even when we restrict the goals to the SECRET and PRECEDES goals in CAPSL. We are investigating general ways to translate goals for future implementation in the translator. In particular, we have found that secrecy goals can be represented with the help of "spell" facts and additional rules as described in [26].

3.4. Implementability

Suppose that a protocol has the messages

```
A -> B: {X}pk(A);
B -> A: X;
```

The transition rule for B could be generated mindlessly as:

$$B_0(B), M(\{X\}_{pk(A)}) \longrightarrow B_1(B,X), M(X).$$

One problem with this rule is that B is actually incapable of decrypting the received message to obtain X. That is, the protocol is *unimplementable*.
The CAPSL translator checks whether a protocol is implementable. In doing so, it deduces what the receiver of a message must do to accept the message, and it also determines whether the sender of a message has the necessary data to construct the message.
Recall that in the MSR notation, a state fact $A_i(\mathbf{y})$ has a sequence $\mathbf{y}$ of terms embodying the memory of the process. In particular, y_1 is the principal for which the process is run. Most of the terms in $\mathbf{y}$ are associated implicitly with protocol variables.
A term t is **computable** from $\mathbf{y}$ by A if

1) $t \in \mathbf{y}$ (sometimes it is convenient to treat $\mathbf{y}$ as a set), or
2) $t = f(\mathbf{x})$ is a functional term whose arguments are computable from $\mathbf{y}$ by A;

and in the second case we check that if f is private, then $x_1 = A$.
Thus, sk(B) is computable by B from $\{B\}$ but not by $A \neq B$.
A message $M(t)$ can be sent from a state $A_i(\mathbf{y})$ if t is computable by y_1 from $\mathbf{y}$ plus any nonces generated in the state transition rule. If $\mathbf{z}$ is the sequence of those nonces, the next state is $A_{i+1}(\mathbf{y}, \mathbf{z})$.
Receivability is more complicated. When a message is encrypted, the receiver must also be able to decrypt it. This is where the INVERT axioms for encryption and other operations come in.
Consider a current state $A_i(\mathbf{y})$ and let $y_1 = A$. A message term m is **receivable** if m is a variable, or m is computable by A, or $m = f(\mathbf{x})$ and for each x_j, either x_j is computable by A or

1) INVERT $f(\mathbf{x}) : x_j \mid \mathbf{w}$ and

2) $\mathbf{w}$ is computable by A, and

3) x_j is receivable.

In each instance of the last case, x_j is *learned* by A, and if the sequence of all learned terms is $\mathbf{z}$, the next state is $A_{i+1}(\mathbf{y}, \mathbf{z})$.
The definition above does not allow a message to be receivable if some message fields must be learned before decrypting other fields. For example, the message `A -> B: K, {X}K;` would not be receivable if K was not already held by B. We can handle this message by rewriting it as a pair of messages, with content K and $\{X\}K$ respectively. In fact, this is unnecessary because CAPSL uses a modified receivability algorithm that acts as though such a rewriting had been done. Our algorithm does not presently allow for fields to be sent in reverse order, e.g., `A -> B: {X}K,K;`.

3.5. Connectors

Connectors translate from CIL to some input representation needed by a protocol analysis tool. Connectors have been written for PVS (SRI's verification environment, used for inductive protocol verification [28, 32], Maude [10], Athena [24], and the NRL Protocol Analyzer. The connectors we have written have been in Java, and make use of common connector support classes for parsing CIL and maintaining an internal tree-structured data representation.

4. Secure multicast

Protocols for secure group management are essential in applications that are concerned with confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership and access control. A variety of new protocols and frameworks have been designed to create multicast groups on a network and support secure group communication (e.g., GDOI [3], GSAKMP [17]. Some existing key exchange protocols for secure communication have been extended to the group setting (e.g., Group Diffie-Hellman GDH [35] and its authenticated form A-GDH [1].
There have been only a few results on the formal analysis of group management protocols (e.g., Pereira and Quisquater analyzed A-GDH [31] and Meadows discovered security flaws in early versions of GDOI [23]. The analysis of group management protocols poses new challenges for formal analysis techniques. New language features and models are necessary to appropriately capture the concepts of such protocols. Moreover, analysis techniques and tools have to be revised and extended.
Multicast CAPSL (MuCAPSL) is an extension of CAPSL affecting all aspects of the environment, from the language and underlying model to analysis techniques and tools. MuCAPSL and its supporting tools are in an early stage of development.

4.1. New MuCAPSL language features

MuCAPSL permits the specification of protocols for secure multicast. The language includes features such as a high-level organization of protocols into suites, a separation of roles for each agent within a protocol, group attributes to capture modifiable persistent state information of group members, and variable-length data structures such as arrays and sequences that are being used as fields in messages or state variables of agents.
In a group setting an agent usually engages in a variety of protocols: to initially set up the group, to distribute new group keys, and to add or delete members. Protocols that conceptually belong together are placed in a *protocol suite*. Within a protocol suite several protocols, typespecs, or environments can be specified. All declarations on the top level of a protocol suite apply to all protocols within the suite. We also refer to the protocols within a suite as *tasks*. A typical suite has the form:

```
SUITE MyGroupMgmt;

TYPESPEC MyGroup;
TYPE MyGroupAgent: GroupMember;
...
END;

PROTOCOL KeyDist;
...
END KeyDist;

PROTOCOL AddMember;
...
End AddMember;

...

END MyGroupMgmt;
```

In the typespec MyGroup we define the type MyGroupAgent for group members of our example. This type is

a subtype of the more general type GroupMember for specifying members of groups.

In a CAPSL typespec we can define principal subtypes and associate immutable or long-term data with a principal by declaring functions. Transient data of principals, associated with a session, is specified via protocol variables. In MuCAPSL, group members store mutable, persistent data in so-called *attributes*. Attributes are shared by different sessions of protocols in a suite and persist between protocol sessions. They are specified in typespecs of group members. There are two functions associated with a group member that jointly serve as a unique identifier: *owner* of type Principal to identify the associated principal (e.g., "Alice"), and *gid* of type Group to identify the associated group (e.g., "Manager" or "Employee"). This way, a principal can be member of several groups. We introduce a new type GroupMember as the default type for group members in the following specification:

```
TYPESPEC GROUPMEMBER;
TYPE GroupMember;
FUNCTIONS
   owner(GroupMember): Principal;
   gid(GroupMember): Group;
END;
```

Note that we also introduced a type Group to capture group identities. In a user-defined group member typespec, the ranges of the functions may be overwritten to subtypes of Principal and Group, respectively.

We will illustrate the use of type specifications for group members using the following simple group attribute structure. Assume a group consisting of N members $M_1, \ldots M_N$ pairwise sharing long-term symmetric keys. We take advantage of the existing typespec for mutual symmetric key nodes (MSKN) in the CAPSL prelude. It defines a subtype Node of Principal and a function $msk(Node, Node)$ with range Skey. Within a group, members are identified by position number (a natural number), which is a changeable attribute due to members leaving the group or new members joining. At any given time, the leader is the member with position 1. Each member stores a short-term group key K_g, addresses of all group members M_{bs} (defined as an array type, a new parameterized type in the MuCAPSL prelude), and current group size N. Here is the full typespec:

```
TYPESPEC MyGroup;
TYPE MyGroupAgent: GroupMember;
FUNCTIONS owner(MyGroupAgent): Node;
ATTRIBUTES(MyGroupAgent);
   Pos: Nat;
   Kg: Skey, CRYPTO;
   Mbs: Array[Principal];
   N: Nat;
END;
```

Attributes are associated with a group member. The type of group member is specified after the ATTRIBUTES keyword, and should be one of the types (if there are more than one) declared in the typespec. Attributes are like protocol variables because their values may change during execution of protocols, but they are different because a group member state always has values for all of the attributes, though initially, some of them may have an explicit "undefined" value.

We illustrate role-based task specifications of multicast protocols with the help of the key distribution protocol. The leader of the group initiates the key distribution protocol whenever a member has been added to or deleted from the group. We distinguish two main roles in the key distribution: the role of the leader M_1 and the role of other members of the group M_i.

Figure 3 roughly illustrates the message flow of the agent in role M_1. M_1 broadcasts the new group key to the entire group (illustrated in Fig. 3 by the square around the role M_i. A unicast message to a member in role M_i would be depicted by leaving the square out). The member uses a sequence field (denoted by $< \ldots >$) that includes $N-1$ copies of the new group key, each encrypted with one of the shared keys. The other group members acknowledge the receipt of the group key by each sending a message that contains their position and a nonce encrypted with the group key. The leader collects all responses. We do not specify a specific role from which the leader receives the responses (the lower arrow is not connected to a sending role). This is done intentionally since the leader is not able to reliably check from the addresses who was sending the message since those addresses are easy to fake. The iteration is indicated by a square that contains the condition $i \in 2..N$ expressing that M_1 receives messages of the form $i, \{Nc_i\}K_g$ until it has collected one for each i (possibly out of order).

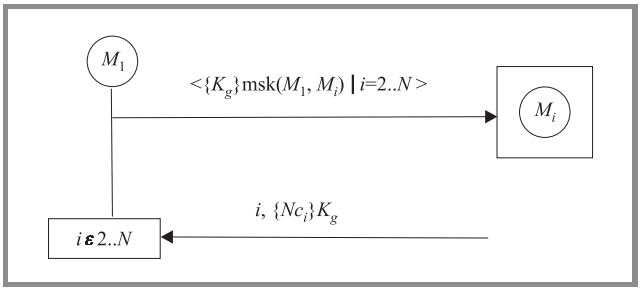

Fig. 3. Key distribution protocol – role M_1.

The separation of roles is also reflected in the protocol specifications. The following shows part of the key distribution protocol, with two roles. The group member playing a role is referred to by the variable implicitly declared in the ROLE statement. The associated principal and group could be derived from this variable via the functions *owner* and *gid*:

```
PROTOCOL KeyDist;

ROLE M1: MyGroupAgent;
VARIABLES i: Nat, FREE;
```

```
          Nc: Array[Nonce];
ASSUMPTIONS Pos=1;
            owner(M1)=Mbs(1);
MESSAGES
  New Kg;
  -> : <{Kg}msk(Mbs(1),Mbs(i))|i=2..N>;
  FOR i IN 2..N DO
      <- : i, {Nc(i)}Kg;
  OD;
END;

ROLE Mi: MyGroupAgent;
ASSUMPTIONS Pos > 1;
MESSAGES ...
END;

END KeyDist;
```

Note that the above protocol specification does not refer to the specific senders or receivers of multicast messages. The sender is implicitly the principal playing the role, and the receiver of a multicast message is implicitly the whole group. Recipients of unicast messages can be specified.
The scope of protocol variables that are defined with the attribute FREE is the statement in which the variable is bound to a range of values. For instance, in the statement

```
FOR i IN 2..N DO
  <- : i, {Nc(i)}Kg;
OD;
```

the scope of *i* is the FOR-loop. Agents do not store the values of free variables in their protocol state or member state.
Attributes of the principal playing the role, such as K_g, can be modified within the protocol. For notational convenience, we refer to attributes in the short form `Kg` instead of the more comprehensive form `Kg(M1)`, since we know that the attributes are associated with `M1`. Note that we need a "New" operator to generate new nonce values. Constructs to express loops or other iterative behavior are necessary to deal with dynamically changing group membership or the need to combine responses from a multicast.
Language features that have not been presented in the current example but have been identified as necessary in the design of MuCAPSL include a syntactical distinction for assignment and tests of group attributes, new built-in cryptographic operators such as a group version of Diffie-Hellman encryption, secret sharing, and threshold encryption.

4.2. New MuCIL features

Hand in hand with the extension of the language goes the extension of the underlying semantic model CIL. In MuCIL new "customized" facts for group member states, protocol states, and multicast messages are introduced. All facts are boolean predicates defined in a functional way that assert either the presence of a group member in the network (*member*(...)) or the existence of an agent in a specific state of a protocol (*state*(...)) or the presence of a multicast message in the network (*mmsg*(...)).
The group member state fact looks as follows:

```
member(owner,gid,terms(attributes))
```

The first parameter refers to the principal that is the member of the group (*owner*), the second parameter identifies the group (*gid*). The third parameter is the list of attributes that is specified for the particular group agent. For the above defined group member type MyGroupAgent, the group member state fact is

```
member(M,G,terms(Pos,Kg,Mbs,N)),
```

with variables of the following types: M : *Principal*, G : *Group*, Pos : *Nat*, Kg : *Skey*, Mbs : *Array(Principal)*, and N : *Nat*.
The protocol state fact of CIL is also extended by a reference to the group identity. Moreover, since there may be several protocols in a suite, the role identifier is composed of the role variable name and the protocol identifier. Thus, a state fact for the group member in role M_1 of the key distribution protocol looks as follows:

```
state(M1,G,roleM1KeyDist,i,terms(...)),
```

with a variable $M1$: *Principal*.
The multicast message fact `mmsg(m)` is simplified compared to CIL since the only parameter it holds is the message content. This is motivated by the fact that sender and receiver addresses in messages do not make a difference from the viewpoint of security analysis since an active attacker can always change addresses. Nevertheless, it may be useful to have this information for purposes such as generating prototypes and the like.
A typical (conditional) rewrite rule in MuCIL contains a member state fact, a protocol state fact and a multicast message fact on both sides. While MSR rules are normally interpreted to delete the left-side facts from the multiset, for MuCIL our convention is to retain message facts implicitly (without repeating them on the right) since they are usually multicast. This is not a real difference, because the attacker can duplicate messages anyway.
The first action and the multicast message of role M_1 in the key distribution protocol are represented by the following MuCIL statement:

```
member(M1,G,terms(1,_,Mbs,N)),
state(M1,G,roleM1KeyDist,0,terms()) ⟶
(∃ Kg) member(M1,G,terms(1,Kg,Mbs,N)),
state(M1,G,roleM1KeyDist,1,terms()),
mmsg(map(lambda(X,ped(msk(Mbs(1),X),Kg)),
          proj(Mbs,2,N)))
```

We do not have a full MSR-style version of MuCIL, but in the above rule we have used some symbols like ⟶ for readability that would not appear in the pure functional form of MuCIL. The underscore in the member fact refers

to a possibly undefined variable, and the actual MuCIL would put in a new variable identifier of an extended type that permits the undefined value.
The rule states that the position of the agent needs to be 1, and the group size and the array variable `Mbs` that holds the other member's addresses need to be defined. The higher-order map and lambda constructs are typical functional language constructs as found, for example, in ML [29]. $lambda(x,u)$ denotes the function mapping x to u, and $map(f,l)$ returns the list of all elements $f(x)$, where x ranges over the elements of l. The projection operator `proj` and the lambda operator, in conjunction with the map operator, allows us to define an array whose elements are K_g encrypted with each member of the sequence of shared keys.
An instance of a multiset of facts is depicted in Fig. 4.

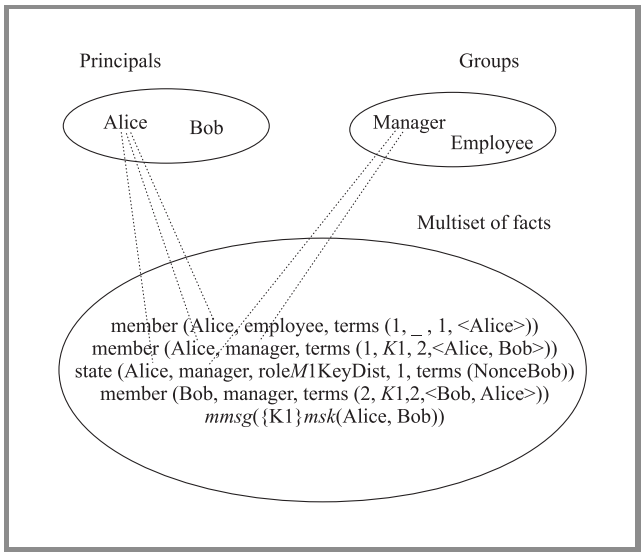

Fig. 4. A multiset of facts.

4.3. Analysing multicast protocols – preliminary results

We have some preliminary experimentation results in using the Maude model checker for the group Diffie-Hellman protocol (GDH) [35], an extension of the Diffie-Hellman key agreement scheme to an arbitrary group size. The authors of that paper suggest three different protocols that are each optimized with respect to certain protocol complexity such as number of rounds, number of messages, sizes of messages, etc. We analyzed the key distribution protocol GDH.2 as an example since it incorporates unicast messages addressed to a particular agent as well as multicast messages addressed to the group.
The group key in GDH.2 is computed from contributions of each group member. For this purpose, each agent has a nonce N_i. The group key is the exponentiation base a raised to the product of all nonces $\Pi_{i=1..n}N_i$ of group members. The exponentiation base is known to every agent, whereas the individual nonces are secret to the particular group members. In a message exchange, agents communicate partial key values, that keep their secret and still allow other group members to compute a shared group key.
Agents that engage in a GDH protocol are identified by a natural number i. They also keep the current group size n. In GDH.2 we distinguish three roles: M_1, M_i, M_n. M_1 is the role of the group member who initiates the key distribution. This group member is characterized by the identification number 1. The agent in role M_n is the "last" member of the group, the one whose identification number equals the group size. All other members are agents in role M_i.
Figure 5 illustrates the communication between group members for a group of size 4. The agent in role M_1 sends out an array consisting of the exponentiation base a and a^{N1} to its neighbor M_2 (an agent in role M_i). Every agent in role M_i receives such an array, multiplies each array element with its own nonce as well as copies the last array element of the received message in its outgoing message. This way, the length of arrays sent between group members always equals the identification number of the receiving agent. This "upflow" phase of GDH.2 consists of unicast messages. Finally, the last group member receives an array of length n from which it computes the group key by raising the last array element to the power of Nn. Moreover, M_n replies in a multicast to the group with an array of partial key values ("downflow" phase) that include its nonce Nn. The other group member can compute the group key from this multicast message by raising the appropriate array element to the power of their nonce.

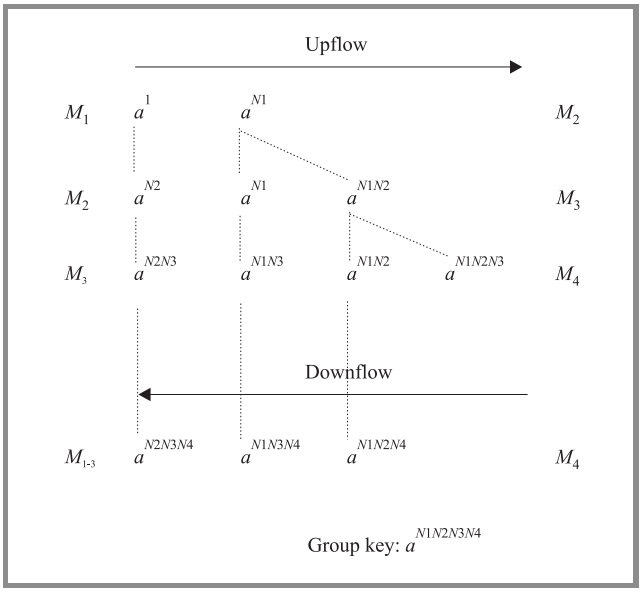

Fig. 5. Overview: group Diffie-Hellman key distribution.

The intent of this protocol is that all group members share a group key. We specified GDH.2 using MuCAPSL and manually translated MuCAPSL into MuCIL. The MuCIL representation was the basis for an analysis using the Maude model checker. In order to deal with MuCIL representations we added arrays, lambda-expressions and undefined

values support to the Maude model checker [10, 14]. The Maude model checker has a strategy for state space exploration that computes all possible runs of the protocol for a given initial state. Since most protocols have an infinite state space, Maude's search strategy has a user-definable parameter to limit the state space investigated. The strategy searches for states that violate any of the defined goals. In the case of GDH.2 we declared the following goal of the protocol: all group members agree on the group key after they have finished a run of the key distribution protocol.

We implemented a limited attacker that has the capacity of misdelivery but does not reconstruct messages. In particular, it is possible that messages are delayed, not delivered or delivered several times. We found a state where the leader got the wrong group key. The attack can be generalized to all group members. This is due to ambiguity in the format of GDH.2 messages. The attack is illustrated in Fig. 6 (for a three-member group).

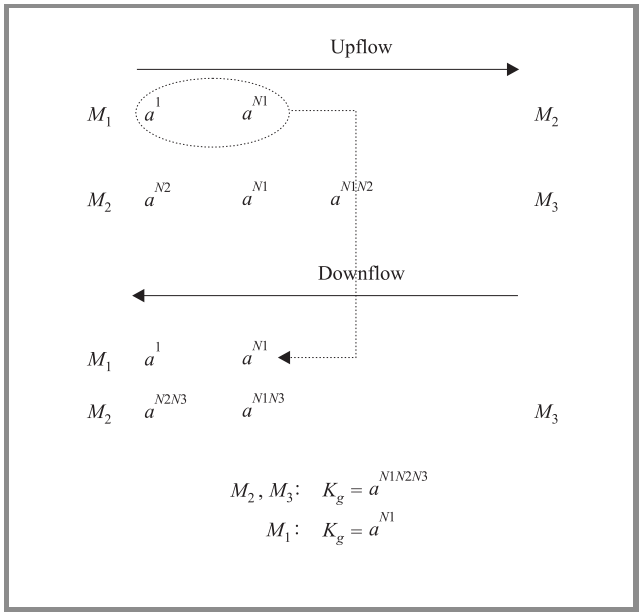

Fig. 6. An attack for GDH.2.

GDH.2 as proposed in [35] does not specify whether group members check the content of messages they receive. In fact, in a correct protocol run, a group member cannot decide from its local memory whether a received message has the right content. In the attack the message that member M_1 sent out in the upflow, will be delivered to M_1 in the downflow. Assuming that the agents do not check the content of the message, M_1 computes the group key a^{N_1}. The other members receive and send messages as specified in GDH.2 and therefore compute the group key $a^{N_1N_2N_3}$. Group member M_1 has not only been tricked into accepting a wrong group key, but also it uses a group key that is known to the attacker. We would like to stress that GDH.2 was not designed to defeat an active attacker.

5. 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 transportable 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.

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. CAPSL clarifies 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. This was done by applying the general type specification mechanism, together with the novel concepts of private functions and invertibility axioms. In the MSR model, session data is held in state memory.

We have begun to broaden the applicability of CAPSL further with the extension to MuCAPSL for multicast protocols. Protocols that conceptually belong together are grouped into protocol suites. Separation of roles within a protocol was introduced to deal with the concurrent asynchronous operation of protocol processes due to multicast transmission and responses. Message handling within groups is supported by new language constructs for iteration and variable-length data structures.

MuCAPSL is built upon the concepts of CAPSL for type specifications. We added attributes for mutable persistent data. Sequence and array type specifications come along with the new computational operators. On the semantic level, MuCIL has group-member state memory to hold mutable persistent group state attributes. Another extension in MuCIL is the role identifier that uniquely identifies the task and protocol.

The intermediate languages CIL and MuCIL were 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. We have developed techniques for inductive protocol proof using PVS and model checking using Maude. In our experiments, we have confirmed that CIL output is a good match for the specification needs of these tools. We are currently investigating security goals for multicast protocols, and we are also developing analysis techniques and tools for MuCAPSL.

References

[1] G. Ateniese, M. Steiner, and G. Tsudik, "New multi-party authentication services and key agreement protocols", *IEEE J. Selec. Areas Commun.*, vol. 18, no. 4, pp. 628–639, 2000.

[2] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[3] M. Baugher, T. Hardjono, H. Harney, and B. Weis, "The group domain of interpretation". Internet Draft, IETF, 2001. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-msec-gdoi-01.txt.

[4] P. Bieber, "A logic of communication in a hostile environment", in *Proc. of the Computer Security Foundations Workshop (III)*. IEEE Computer Society Press, 1990, pp. 14–22.

[5] D. Bolignano, "An approach to the formal verification of cryptographic protocols", in *3rd ACM Conference on Computer & Communication Security*. ACM Press, 1996, pp. 106–118.

[6] S. Brackin, "An interface specification language for automatically analyzing cryptographic protocols", in *Symposium on Network and Distributed System Security*. Internet Society, Febr. 1997.

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

[8] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "A meta-notation for protocol analysis", in *12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1999, pp. 55–69.

[9] E. Clarke, S. Jha, and W. Marrero, "Using state space exploration and a natural deduction style message derivation engine to verify security protocols", in *Proc. IFIP Work. Conf. Program. Concepts Meth. (PROCOMET)*, 1998.

[10] G. Denker, "Design of a CIL connector to Maude", in *Workshop on Formal Methods and Computer Security*, H. Veith, N. Heintze, and E. Clarke, Eds. Carnegie Mellon University, July 2000.

[11] G. Denker and J. Millen, "CAPSL integrated protocol environment", in *DARPA Information Survivability Conference (DISCEX 2000)*. IEEE Computer Society, 2000, pp. 207–221.

[12] G. Denker, J. Millen, J. Kuester-Filipe, and A. Grau, "Optimizing protocol rewrite rules of CIL specifications", in *13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.

[13] G. Denker, J. Millen, and H. Ruess, "The CAPSL integrated protocol environment". Tech. Rep. SRI-CSL-2000-02, SRI International, 2000.

[14] G. Denker, J. Meseguer, and C. Talcott, "Protocol specification and analysis in Maude", in *Form. Meth. Secur. Protoc.*, 1998, LICS '98 Workshop.

[15] D. Dolev and A. Yao, "On the security of public key protocols", *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 198–208, 1983 (also STAN-CS-81-854, May 1981, Stanford U.).

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

[17] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer, "Group secure association key management protocol". Internet Draft, IETF, 2001. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-msec-gsakmp-sec-00.txt.

[18] R. Kemmerer, "Analyzing encryption protocols using formal verification techniques", *IEEE J. Selec. Areas Commun.*, vol. 7, no. 4, 1989.

[19] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *Proceedings of TACAS*, *Lecture Notes in Computer Science*. Springer, 1996, vol. 1055, pp. 147–166.

[20] G. Lowe, "Casper: a compiler for the analysis of security protocols", *J. Comput. Secur.*, vol. 6, no. 1, pp. 53–84, 1998.

[21] J. Millen, S. Clark, and S. Freedman, "The Interrogator: protocol security analysis", *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 2, pp. 274–288, 1987.

[22] C. Meadows, "A system for the specification and verification of key management protocols", in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1991, pp. 182–195.

[23] C. Meadows, "Experiences in the formal analysis of the GDOI protocol". Slides, Dagstuhl Seminar Specification and Analysis of Secure Cryptographic Protocols, 2001. [Online]. Available: http://www.informatik.uni-freiburg.de/ accorsi/dagstuhl.

[24] J. Millen, "A CAPSL connector to Athena", in *Workshop of Formal Methods and Computer Security*, H. Veith, N. Heintze, and E. Clarke, Eds. CAV, 2000.

[25] J. Millen, "CAPSL web site". 2001. [Online]. Available: http://www.csl.sri.com/ millen/capsl.

[26] J. Millen and H. Rueß, "Protocol-independent secrecy", in *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[27] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers", *Commun. ACM*, vol. 21, no. 12, pp. 993–998, 1978.

[28] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "PVS: an experience report", in *Applied Formal Methods – FM-Trends'98*, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Eds., *LNCS*. Springer, 1998, vol. 1641, pp. 338–345. [Online]. Available: http://www.csl.sri.com/papers/fmtrends98.

[29] L. C. Paulson, *ML for the Working Programmer*. 2nd ed. Cambridge University Press, 1996.

[30] L. Paulson, "The inductive approach to verifying cryptographic protocols", *J. Comput. Secur.*, vol. 6, no. 1, pp. 85–128, 1998.

[31] O. Pereira and J. Quisquater, "A security analysis of the cliques protocol suites", in *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 73–81.

[32] H. Rueß and J. Millen, "Local secrecy for state-based models", in *Form. Meth. Comput. Secur., CAV Worksh.*, Chicago, IL, July 2000.

[33] A. W. Roscoe, "Modelling and verifying key-exchange protocols using CSP and FDR", in *8th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1995, pp. 98–107.

[34] S. Schneider, "Verifying authentication protocols in CSP", *IEEE Trans. Softw. Eng.*, vol. 24, no. 9, pp. 741–758, 1998.

[35] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-Hellman key distribution extended to group communication', in *Proc. 3rd ACM Conf. Comput. Commun. Secur.*, 1996.

Jonathan K. Millen is a Senior Computer Scientist in the Computer Science Laboratory at SRI International, working in the area of dependable systems and cryptographic protocol verification. From 1969 to 1997 he worked at the MITRE Corporation. He is co-Editor-in-Chief of the *Journal of Computer Security*, an associate editor of the *ACM Transactions on Information and System Security*, and the founder of the annual *IEEE Computer Security Foundations Workshop*. He holds a Ph.D. in mathematics from Rensselaer Polytechnic Institute, an M.S. from Stanford, and an A.B. from Harvard.

e-mail: millen@csl.sri.com
SRI International
Room EL 233
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Grit Denker is a Computer Scientist in the Computer Science Laboratory at SRI International in California. From 1995 to 1996 she worked as an assistant professor at the Technical University of Braunschweig in Germany. Her main areas of interest are formal specification and verification of cryptographic security protocols, security for the Semantic Web, and logic-based approaches for distributed system analysis. She holds a Ph.D. in computer science and an M.S. in mathematics from the Technical University of Braunschweig.

e-mail: denker@csl.sri.com
SRI International
Room EL 284
333 Ravenswood Avenue
Menlo Park, CA 94025, USA