THE NRL PROTOCOL ANALYZER: AN OVERVIEW

CATHERINE MEADOWS

▷

The NRL Protocol Analyzer is a prototype special-purpose verification tool, written in Prolog, that has been developed for the analysis of cryptographic protocols that are used to authenticate principals and services and distribute keys in a network. In this paper we give an overview of how the Analyzer works and describe its achievements so far. We also show how our use of the Prolog language benefited us in the design and implementation of the Analyzer. ◁

1. INTRODUCTION

A cryptographic protocol is a communication protocol that uses encryption in order to achieve goals such as distribution of cryptographic keys or authentication of principals and services, over a network that may contain a number of hostile intruders who may be actively trying to subvert the goals of the protocol. For example, if the protocol is intended for key distribution, the intruder may attempt to discover the session key, or more subtly, attempt to convince principals that some other word chosen by the intruder is itself the session key. If the protocol is intended for authentication of principals, the intruder may attempt to pass itself off as some other principal.

Most cryptographic protocols are designed to function under very adverse conditions. In general, it is assumed that the intruder has complete control of all communication channels, and thus can read all traffic, destroy or alter traffic, and generate traffic of its own. It is also usually assumed that some principals are cooperating with the intruder, and thus that the intruder will be able to perform operations such as encryption that are available to honest users of the network.

Given such requirements, it is not surprising that it is difficult to design cryptographic protocols that are free of flaws. Even quite simple protocols have been

This paper is an expanded version of the paper "The NRL Protocol Analyzer: An Overview", published in *The Proceedings Second International Conference on the Practical Applications of Prolog*, April 1994.

Address correspondence to Code 5543, Naval Research Laboratory, Washington, DC 20375

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

discovered to have flaws that in many cases were not discovered until some time after they were published or even implemented. These flaws were independent of the strengths or weakness of the particular cryptoalgorithm used.

As an example of the kinds of flaws that can occur, consider the following authentication protocol, originally proposed as part of an ISO standard, that makes use of a public-key cryptosystem [5], which is a cryptosystem in which the keys used for encryption and decryption are separate. This allows the encryption key to be distributed widely, and thus anyone can send a message to a party and be sure that it can only be read by its intended recipient by encrypting the message with that party's public key. The private keys can be used not only to decrypt messages, but to verify the origin of a message. A party can "sign" a message by decrypting it with its private key. A recipient verifies the signature by encrypting it with the public key and comparing it with the original message. If they match the recipient knows that only the owner of the public key could have sent the message, since only he could have produced the signature.

In the protocol we are about to examine, two parties A and B wish to be sure that they are communicating with each other. Each party P has a public key K_p and a private key $K_p^{-1}(X)$. The application of public or private key operations to a message M is denoted by K(M), where K is a public or private key. A and B also possess the ability to generate nonces, which are unique random numbers that are to be used only once and then thrown away. The purpose of a nonce is to guarantee that a message is recent. A can guarantee that a message from B is recent by sending B a nonce. B then sends back the message, together with the nonce, signed with his private key. A can now be sure that the message was sent after he sent the nonce.

In the protocol that follows, A and B are using the mechanisms described above to verify that they are communicating with each other. We use the common notation $A \hookrightarrow B$: M to stand for "A sends message M to B."

1. $A \hookrightarrow B$: A, N_a
   where N_a is a nonce, that is, a number chosen by A that has never been used before.
2. $B \hookrightarrow A$: N_b, A, $K_b^{-1}(N_b, N_a, A)$
   where $K_b^{-1}(X)$ denotes the signing of X with B's private key, and where N_b is a nonce chosen by B.
   A then verifies B's signature. A now believes that it has heard from B in response to A's original message, since the message is signed by B and contains A's nonce.
3. $A \hookrightarrow B$: $N_{a'}$, B, $K_a^{-1}(N_{a'}, N_b, B)$
   where $N_{a'}$ is a new nonce generated by A.
   B checks the signature on A's message. B now believes that it has heard from A.

In [6], the following attack is presented:
Let I be the intruder. I_M denotes the intruder impersonating M.

1. $I_A \hookrightarrow B$: A, N_x
   where N_x is a nonce generated by I.
2. $B \hookrightarrow A$: N_b, A, $K_b^{-1}(N_b, N_x, A)$
   The intruder I intercepts this message and prevents it from reaching A.

3. $I_B \hookrightarrow A$: B, N_b.
4. $A \hookrightarrow B$: N_a, B, $K_a^{-1}(N_a, N_b, B)$
   B checks the signature and concludes that A has successfully initiated contact with it and that N_x was a nonce originating from A. A however, does not have a corresponding belief that it is communicating with B.

As we see, in general it is not easy to see whether a cryptographic protocol is secure simply by looking at it; even in a simple protocol such as the one just given, flaws can be very subtle. This has been shown also in a number examples in the literature of protocols that were published, believed to be sound, and later shown to have security flaws [3, 4, 10, 11, 14]. Thus it is necessary to develop some rigorous means of reasoning about cryptographic protocols.

In the last five years there has been a great deal of work done in developing formal models of cryptographic protocols. As in the analysis of conventional communication protocols, there have been two kinds of techniques applied to this problem. One is to use logics of knowledge and belief to model the beliefs that evolve in the course of a protocol. The best known of these is the Burroughs, Abadi, and Needham logic [3]. Another is to model the protocol as an interaction between a set of state machines and to attempt to prove a protocol secure by specifying insecure states and attempting to prove them unreachable, by use either of exhaustive search backwards from the state, or by the use proof techniques for reasoning about state machine models. This is the approach taken by the NRL Protocol Analyzer.

A number of challenges exist that must be met when applying this type of approach to cryptographic protocol analysis. One arises from the fact that the words used in a cryptographic protocol obey certain reduction rules; for example, encryption and decryption with the same key in a single-key cryptosystem cancel each other out. Another arises from the fact that, for all practical purposes, the search space is unbounded. For example, although the set of possible keys is finite, it is large enough so that a key cannot be found by exhaustive search. Thus, for the purpose of the model, we assume that it is infinite. Likewise, if we encrypt a word over and over with a key, we assume that we produce an unbounded set of words. Thus, if we attempt to use search alone, we will not succeed. Some other kinds of proof techniques are necessary.

The NRL Protocol Analyzer was designed specifically to meet these challenges. It uses narrowing to handle the fact that words obey reduction rules, and it includes techniques and automatic support for using induction to prove that infinite sets of states are unreachable. Thus the NRL Protocol Analyzer can be used to prove security properties of cryptographic protocols as well as locate security flaws. The NRL Protocol Analyzer has been successful in doing both. In particular, it has been used to find previously unknown flaws in the Simmons Selective Broadcast Protocol [14] and the Burns-Mitchell Resource Sharing Protocol [2], and some hidden assumptions in the Neuman-Stubblebine reauthentication protocol [13] and the Aziz-Diffie wireless communication protocol [1]. The results of these analyses are contained in [10, 11, 16, 12]. The NRL Protocol Analyzer has also been able to reproduce previously known attacks, including the one described in this paper. In particular, the Analyzer's success in reproducing known attacks is documented in [9], in which several different protocol analysis tools are applied to the same flawed protocol.

The Analyzer is also of interest as a Prolog application. It has been developed

in Prolog from the beginning, and makes extensive use of many of the special properties of Prolog. In this paper, we will describe, not only how the Analyzer operates, but how it makes use of the Prolog language.

The remainder of the paper is organized as follows. Section 2 describes the model and specification language used by the Analzyer. Section 3 describes how the Analyzer works and how the user interacts with the Analyzer. We also give some details about how the Analyzer is implemented in Prolog. In Section 4 we give a brief history of the Analyzer's development. In Section 5 we describe some of the advantages and disadvantages of using Prolog to solve this problem.

2. THE MODEL AND SPECIFICATION LANGUAGE USED BY THE ANALYZER

2.1. Description of the Model and Specification Language

The Analyzer is based upon a version of the term-rewriting model of Dolev and Yao [8]. In the Dolev-Yao model, it is assumed that there is an intruder who is able to read all message traffic, modify and destroy any message traffic, and perform any operation (such as encryption or decryption) that is available to legitimate user of the protocol. However, it is assumed that there is some set of words (for example encryption keys possessed by honest principals, or messages that have been encrypted) that the intruder does not already know. His goal is to find out these words. Since any message received by an honest principal can be thought of as having been sent by the intruder, we can think of the protocol as an algebraic system being manipulated by the intruder. His goal is to manipulate it in such a way that a "secret" word is produced.

The words produced by the algebraic system will obey a set of reduction rules. For example, encryption and decryption with the same key using a private-key algorithm is self-cancelling. Thus, we can think of the intruder as attempting to solve a word problem in a term-rewriting system. Using this insight, Dolev and Yao, and later Dolev, Even and Karp [7], developed a set of algorithms for proving the security of certain limited classes of protocols.

Although our model is based on that of Dolev and Yao, the general approach we take to proving security properties of protocols is somewhat different. For one thing, we extend the goals of the intruder to include more than just finding out secret words. Many protocols (such as the example given in the introduction to this paper) are broken, not by the intruder discovering a secret word, but by the intruder convincing a principal that a word has certain properties that it does not have. For example, a protocol can be broken if the intruder can convince a principal that a word already known by the intruder is a session key. Thus we extended our model to include local state variables possessed by the principals.

The other difference was that we wanted to be able to examine a more general and open-ended class of protocols than those of Dolev and Yao. Thus, instead of developing a set of algorithms, we developed a general procedure for proving security properties of protocols by proving user-specified protocol states unreachable, and an interactive Prolog program that facilitates this procedure.

In the NRL Protocol Analyzer, protocols are specified as a set of transitions of state machines. Each transition rule is specified in terms of the following:

1. words that must be input by the intruder before a rule can fire;
2. values that must be held by local state variables before the rule can fire;
3. words output by the principal (and hence learned by the intruder) after the rule fires, and;
4. new values taken on by local state variables after the rule fires;
5. event statements that describe the transition in terms of words used and principals involved.

Transition rules can describe, not only the actions of an honest user, but the actions of an intruder who produces new messages out of old by performing some operation such as encryption or decryption.

Participants in a protocol communicate by exchanging *words*. We make the assumption that an intruder, upon seeing a word, is able to determine its history and significance, although he may not know the words that were used to construct the word he sees. Thus, if the intruder sees the word X = e(key(user(A)),message(A,N)) we assume that he is aware that X is a message from A generated by a key belonging to A, although he will not know key(user(A)) or message(A,N) unless he has seen them previously. Although in actual fact an intruder may not always have access to this sort of information, most of the protocols we have examined are designed to be secure under such strong assumptions about the level of knowledge available to the intruder. Thus we make these assumptions in order to have the best possible assurance of security.

The assumptions about the knowledge of words available to the honest users are much more restrictive. An honest user will recognize the significance of a word if he has explicitly stored it as the value of a learned fact (see below). We may also specify a list of words that we assume the user is able to recognize, and what it is about them that he is able to recognize. For example, we may assume that a user is able to recognize all usernames as usernames, but may not be able to tell whether or not they belong to honest or dishonest users. Finally, an honest user is always able to recognize one attribute of a word: its length. Thus if a user is expecting a two block long message, and receives one that is one block long, he has the ability to reject it.

The words involved in these rules obey a set of reduction rules. A few of these are built-in rules supplied by the system, but most are described by the specification writer. We make the requirement that the set of reduction rules be congruent and terminating, in order that narrowing algorithms can be applied.

Words are built by applying function symbols recursively to atoms or variables. Both function symbols and atoms must be specified by the specification writer. Some function symbols, such as the ones specifying encryption and decryption, stand for operations on words that can be performed by any principal who has the capability and who knows the arguments the symbol is being applied to. These function symbols are denoted *operations* in a protocol specification. If the intruder is assumed to be able to perform an operation, this must be indicated in its specification.

Each protocol participant possesses a counter that is incremented every time a protocol rule is fired that either changes its set of learned facts or causes it to produce a word or words. The intruder also possesses a counter. This counter is incremented every time the intruder performs an action that involves communication with itself, such as occurs for example when the intruder performs encryption.

If A is an honest participant in a protocol, we define the *set of runs of the protocol local to A* to be a set of sequences of state changes 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 that is changed during that run is identified by the run's identifier. Any possible application of a protocol rule must result in a state change that belongs to some run. For each possible local run, there is a protocol rule describing its beginning and a protocol rule describing its ending. We note that a participant may participate in two or more runs concurrently.

Each honest protocol participant possesses a set of local state variables, that we call learned facts, or lfacts. Each lfact is relevant to a given run of the protocol. An lfact is described using an *lfact function*, which has four arguments. The first identifies the participant A who knows the fact. The second identifies the run of the protocol. (If A is the intruder, then this argument will always be set equal to the value of the intruder's counter at that time.) The third indicates the nature of the fact, and the fourth gives the present value of A's counter. The value of the lfact is either a list of words that make up the content of the fact, or if the fact does not have any content, it is "[]", the empty list. Thus, for example, suppose that user(A) attempts to initiate a conversation with user(B) during local round N at time T. This can be expressed by the learned fact

$$\text{lfact(user(A),N,init_conv,T)} = [\text{user(B)}]$$

At any time S prior to T, the value of the lfact would be "[]". If user(B) receives a message X during local round S at time P which is apparently from user(A) attempting to initiate a conversation, this can be expressed by the learned fact

$$\text{lfact(user(B),S,rcvd_init_con,P)} = [\text{user(A),X}]$$

The value of a learned fact is calculated in the following way. First, if we have lfact(A,B,C,X) = Y, that is, the value of the lfact is Y when A's local counter is set to X. Suppose now that a rule fires that causes A to either generate a message or to change one of its lfacts, or both. Then A's local counter is set to s(X). If the rule causes lfact(A,B,C,X) to be set to Z, we have lfact(A,B,C,s(X)) = Z. If the rule causes no change to the lfact, then we have lfact(A,B,C,s(X)) = Y. All lfacts are initially empty, that is, set equal to the empty list [].

Besides producing words and learned facts, each protocol rule also produces an *event function*. The event function gives a description of the event that occurred when that rule fired. Since these are used only to identify a transition, not to specify how it behaves, we do not go into any more detail about them here.

Transition rules engaged in by the intruder, unlike those engaged in by honest participants, are not specified explicitly. Instead, they are constructed from specifications of operations that the intruder is indicated as being able to perform. From each such specification a transition is constructed in which the arguments of the operation are input as words the intruder must know, and the output is the result of performing the operation. These transitions can be displayed by querying the Analyzer.

All transition rules are stored as Prolog facts whose arguments are words input, lfacts input, words output, and lfacts output, as well as some other relevant information about the rule, such as the corresponding event function. When a variable

is used in the specification of a rule, it is stored as such in its Prolog representation. When a rule is applied, the variables in the rule are instantiated to the appropriate values. How this is done is described in more detail in Section 3.

2.2. An Example Specification

In this section, we give an example of how the ISO protocol we described above would be specified using the techniques described above.

We begin our specification by listing the words and operations. The words involved are user names, random numbers, and private and public keys. The operations are the built-in list manipulation and id_check operations, and public and private key encryption. (We note that only private key encryption is used in the messages, but public key encryption is used to verify the signatures.)

We begin with keys first. There are two types of keys, public keys and private keys, one each for each user. Thus we denote keys by pubkey(A) and privkey(A), where A is a user name.

We now look at names. We note that there are two types of users, honest ones, and dishonest ones. The honest ones follow the rules of the protocol, while, according to our model, the dishonest ones are in league with the intruder, and thus are willing to share any information that they know with the intruder. Thus the transition rules describing the protocol will describe only the behavior of honest users. The actions of dishonest users (e.g., sending messages or performing operatings) are subsumed by the actions of the intruder; that is, if a dishonest user performs an actions, we can represent it by an intruder action. Moreover, any word (such as a key) known by a dishonest user is assumed to be known by the intruder. Because of this, we need to distinguish between honest and dishonest users. We do this by denoting them as user(A,dishonest) and user(A,honest), where A is a variable standing for some symbol distinguishing two honest users from each other or two dishonest users from each other.

Finally, we look at random numbers. We want random numbers generated by different individuals at different times to be unique, so we give each number the name of the originator and the time at which it was originated as argument. Thus each random number is designated as rand(user(A,X),N) where user(A,X) is a user name and N is an integer.

We next look at operations. Instead of using two different symbols for public key encryption and decryption, we use the same symbol for encryption and decryption and distinguish between the two by specifying whether a public or private key is to be used. We assume that the intruder is able to perform public key encryption if he has access to the keys.

We have the option of assigning lengths to words. Since we do not have any information on the lengths of the words used in the protocol, we do not take advantage of it, except to assign length zero to words that do not stand for bit strings sent over the network. We now have four function symbol descriptions, one for user names, one for public keys, one for private keys, one for random numbers, and one operation description for public key encryption. We also have two atoms, "honest", and "dishonest." Note that we have not specified any atoms that instantiate numbers or user ids, since we do not need to instantiate these in our analysis.

Our specification of function symbols, operations, and atoms is as follows.

```
op(1):pke(X,Y) --> Z::length(Z) = length(Y):pen.

fs(1):user(A,H) --> U::.
fs(2):rand(user(A,H),N) --> R::.
fs(3):privkey(user(A,H)) --> K::.
fs(4):pubkey(user(A,H)) --> K::.

atom(1):honest --> H:0.
atom(2):dishonest --> H:0.
```

We next include two rewrite rules. They say that public key encryption first with a public key, and then with a corresponding private key, and vice versa is self-cancelling.

```
rr1: pke(privkey(X),pke(pubkey(X),Y)) => Y.
rr2: pke(pubkey(X),pke(privkey(X),Y)) => Y.
```

Next we include a list of the words known by the intruder. We assume that the intruder knows all user names and all public keys. We also assume that he knows everything known by a dishonest user, including all random numbers generated by dishonest users, and all private keys belonging to dishonest users. We thus have:

```
known: user(A,X), rand(user(A,dishonest),N),
privkey(user(A,dishonest)), pubkey(user(A,X)).
```

We are now ready to write the specification of the protocol itself. For reasons of space, we limit ourselves to describing the behavior of the principal initiating the protocol. The specification of the behavior of the other party is similar. The initiator first sends off a message containing the name of who he wants to talk to and a nonce, and stores both values in a state variable called init_nonce1.

```
rule(1)
If:
count(user(A,honest)) = [N],
then:
count(user(A,honest)) = [s(N)],
intruderlearns([user(A,honest),rand(user(A,honest),N)]),
lfact(user(A,honest),N,init_nonce1,s(N)) =
   [user(B,W),rand(user(A,honest),N)],
EVENT:
event(user(A,honest),N,init_request,s(N)) =
   [user(B,W),rand(user(A,honest),N)].
```

The next two transitions describe the behavior of an initiator user(A,honest) upon receiving what purports to be a response to his initiation message. We need two transitions, one describing user(A,honest) receiving the message and checking it, and the other describing what user(A,honest) does if the check succeeds. The check is described by the use of the built_in id_check function on the result of applying user(B,W)'s public key to the signature and on the result of concatenating Y, user(A,honest))'s stored nonce Z and (user(A,honest))'s name. Id_check(X,X) reduces to "ok", and so can be used to describe the check for identity.

```
rule(2)
If:
count(user(A,honest)) = [M],
intruderknows([Y,user(A,honest),Z]),
lfact(user(A,honest),N,init_nonce1,M) = [user(B,W),X],
lfact(user(A,honest),N,init_gotnonce,M) = [],
then:
count(user(A,honest)) = [s(M)],
lfact(user(A,honest),N,init_gotnonce,s(M)) = [user(B,W),Y,
id_check(pke(pubkey(user(B,W)),Z),(Y,X,user(A,honest)))],
EVENT:
event(user(A,honest),N,init_testnonce,s(M)) = [user(B,W),X,Y,Z].

rule(3)
If:
count(user(A,honest)) = [M],
lfact(user(A,honest),N,init_gotnonce,M) = [user(B,W),Y,ok],
lfact(user(A,honest),N,init_nonce1,M) = [user(B,W),X],
lfact(user(A,honest),N,init_nonce2,M) = [],
then:
count(user(A,honest)) = [s(M)],
intruderlearns([rand(user(A,honest),M),user(B,W),
pke(privkey(user(A,honest)),(rand(user(A,honest),M),Y,user(B,W)))]),
lfact(user(A,honest),N,init_nonce2,s(M)) = [rand(user(A,honest),M)],
EVENT:
event(user(A,honest),N,init_accnonce,s(M)) =
[user(B,W),rand(user(A,honest),M),X,Y].
```

We note that it was necessary to have two transitions here instead of one, because if we collapsed them into one, we would have to have as one of the conditions that id_check on pke(pubkey(user(B,W)),Z) and (Y,X,user(A,honest)) succeed. This would require the use of operations appearing in a rewrite rule (namely "id_check" and "pke") appearing in the "if" part of a rule, which is not allowed.

3. HOW THE PROTOCOL ANALYZER WORKS

3.1. How the Analyzer Finds and Stores States

The user of the Protocol Analyzer queries it by presenting it with a description of a state in terms of words known by the intruder and values of local state variables. All words used in that state specification are assumed already to be in their reduced form. The Analyzer takes each subset of the words and local state variables specified by the user and, for each transition rule, uses a narrowing algorithm to find a complete set of substitutions (if any exist) that make the output of the rule reducible to that subset. In each case when that is done, the input of the rule, together with any portions of the state that were not matched, are displayed as a description of a state that may immediately precede the specified state. Thus the Analyzer gives a complete description of all states that may precede the specified state.

The Protocol Analyzer does not display all the states that it finds. Instead, it uses a "generate and test" strategy. First, a complete description of all possible states that can precede the specified state are generated using the narrowing algorithm. Then, various rules are used to discard states that have been shown to be unreachable, as well as paths that are redundant in the sense that that path is reachable only if some other existing path is reachable. Some of these rules are built into the Analyzer, while others are generated by the user as a result of proving results about the unreachability of classes of states.

Built-in rules include rules for discarding states in which a word is used before it has been generated, rules for discarding paths that are interleaving of other paths, rules for discarding paths that require the intruder to look for the same word twice, and so forth. The user-generated rules are of more interest, and they are what gives the Analyzer much of its power. We describe these in the next section.

Once a state has passed all the tests, it is stored in a database in the following way. Suppose that after querying a state S, we find that a state T precedes σS, where σ is some substitution. Then we store this fact as the clause

success([Numlist,T],[Numlist1,σS]),

where Numlist and Numlist1 give a Dewey decimal-style numbering of the states. Thus if Numlist1 is N1.N2. ... .Nk and T is the M'th state found preceding S, then Numlist is N1.N2. ... .Nk.T. If state Q is later found to precede τT for some τ, we represent this by a clause of the form

success([Numlist2,Q],[Numlist,τT]).

If the user wishes to see a path from Q to S, this is done by unifying the second term of

success([Numlist2,Q],[Numlist,τT])

with the first term of

success([Numlist,T],[Numlist1,σS])

to obtain a path Q, τT, $\tau\sigma$S. On the other hand, if the user wishes to see a path from T to S, the Analyzer will simply display T, σS.

This approach to storing state descriptions has several advantages. For one thing, a user, by looking at different subpaths, can see the unifications being built up before his or her eyes. Secondly, it is easy to "undo" a line of query just by deleting a set of tuples, since all information about previous unifications is still stored in the database.

3.2. Database of Requirements on Reachable States

In many cases the user of the Analyzer may be able to prove that some specified state is unreachable, or that it is reachable only under the condition that the state variables take on certain values. For example, suppose that all session keys generated by a key server are of the form seskey(server,N), and all nonces used to verify freshness of messages are designated by rand(A,M), where in both cases the first argument is the name of the originator, while the second argument is the time (by the originator's local clock) when it was generated. Suppose furthermore that we have a state variable W that designates a word that a principal has accepted as

a session key. Then we may be able to prove, for example, that W can never be of the form rand(A,M), (nonces can't be accepted as session keys), or that W can only be of the form seskey(server,N) (only session keys can be accepted as session keys), or that W can only be of the form rand(A,M) *or* seskey(server,N).

The user can enter such facts into a database so that, whenever the Analyzer encounters a solution state that has been specified as unreachable, it discards it. If the Analyzer encounters a solution state containing a state variable that has been specified only to take on a certain value or values, it attempts to unify the value given in the solution with the value or values given in the database. If it cannot perform any of the unifications, then the solution state is discarded. Thus, if a solution state says that W is an accepted session key, and the database says that only words of the form seskey(server,N) can be accepted session keys, the Analyzer will attempt to unify W with seskey(server,N). If one of the unifications can be made, the state variable will now take on that value. If the database gives two or more choices for conditions on a word, several different solutions will be created, each with the state variable set to the appropriate value. If none of the unifications can be made, the solution state will be discarded.

It is possible to generate rules for the database automatically after having proved that a state is reachable only under certain conditions. One types the command "displaycons" while the search tree is still present in the Analyzer, and the appropriate rule is displayed. The rule is calculated in the following way. First each path to the goal that does not begin in an unreachable state is examined and the substitutions made to the variables in the goal that are produced by that path is collected. The entire collection is then examined for substitutions that are subsumed by others in the collection, and the subsumed substitutions are culled. Finally, the remaining substitutions are displayed in a form readable by the Analyzer. These can now be entered by the user into a file that can be input to the Analyzer.

3.3. Database of Formal Languages

One of the most powerful tools for limiting the search space in the Analyzer is the use of formal languages. Since the set of words generated in our model is infinite, one of the most common sources of infinite loops in the Protocol Analyzer is the case in which a search produces an unbounded set of words to be found. To give a simple example, suppose that we are trying to find out if the intruder can find a word of the form e(k,X), where k is a specific word, X is a variable standing for any word, and e(Y,Z) denotes encryption of Z with Y. Upon asking the system whether or not an intruder can find a word of the form e(k,X), we are told that this is possible only if the intruder can find a pair of words Y and e(Y,e(k,X)). Upon asking the system whether or not an intruder can find an irreducible word of the form e(Y,e(k,X)), we are told that this is possible if and only if he can find Z and e(Z,e(Y,e(k,X))), and so forth. At this point it should become apparent that we are in danger of entering an infinite loop, and that it would be helpful to be able to prove that we are doing so. To this end, for each possible e(k,X), we define a language **F** as follows:

$\mathbf{F} \rightarrow$ e(k,**A**)
$\mathbf{F} \rightarrow$ e(**A**,**F**)

where **A** is the language consisting of all irreducible words.

We want to show that it is impossible to find any irreducible word of **F** unless some other irreducible words of **F** have already been found. [1] We can then conclude that all irreducible words of **F** (including e(k,X)), are unobtainable.

We do this as follows. We begin by creating a list of expressions so that each word of **F** is a case of some such expression. We may also put conditions on the expressions to the effect that certain words in the expression are members of some language. In this case, we have two such expressions with the corresponding conditions:

1. e(k,X)
2. e(A,B) where B is a member of **F**

We now check each expression by running the system on it to determine what words must be known. We already know that, in order to learn e(k,X), the intruder must know a word of **F**, so it remains to check the second expression. In that case, we simply ask the Analyzer how to find the word e(A,B). Suppose that the Analyzer tells us that the only case in which e(A,B) is found is one which requires prior knowledge of e(Z,e(A,B)) for some Z. Since e(A,B) is a word of **F**, so is e(Z,e(A,B)), and we are done.

It is up to the user to specify languages, but membership in a language and the unreachability of a language are proved by the Analyzer. The user must obey certain restrictions in specifying languages; in particular we make the restriction that the language be context-free, in the sense that every production in a language is of the form

F $\rightarrow$ X, Conditions

where X is some expression and Conditions is a set of conditions on X, such as its length, or the fact that it is not equal to some other word. This allows us to represent each production in the language as a Horn clause and to use Prolog to check for language membership in a natural way. The translation is done as follows. Each terminal symbol is assigned a constant name. Each nonterminal in X is replaced by a variable. If the nonterminal does not stand for the language of all words, then the condition langmember(V,Symbolname) is added to Conditions, where V is the variable substituting for the symbol and Symbolname is the name of the symbol. Likewise, a variable W is substituted **F** and a name Fname is assigned to it. The clause

```
languagerule(W,Fname,Conditions)
```

is stored in the database.

3.3.1. Procedure for Proving a Language Unreachable Our goal is to show that, if the intruder knows a word A, and A is a member of a language L, and W is the set of words that the intruder knows before finding A, then W must contain a member of L. We do this by showing that, whenever langmember(σA,L) can be shown for some substitution σ, then langmember(X,L) holds for some X in σW.

In order to make this possible, we introduce a procedure

[1] We restrict ourselves to irreducible words because all words learned by the intruder are assumed to be in irreducible form.

expandconditions(+*C,-D,+Type)

where expandconditions(C,D,allsubs), given a condition C as input, finds a substitution σ and a condition D implying σC, and expandconditions(C,D,always) finds D the disjunction of all conditions implying C. This first is calculated by substituting for conditions of the form langmember(X,L) executions of proofs using unification, while the second is calculated by substituting executions of proofs using subsumption. This is done as follows:

For each occurrence of langmember(X,L) in a condition C, where X is not a variable and L is not "all", expandconditions(C,D,allsubs) first finds a rule languagerule(test,N,languagerule(A,L,V),C1) and a most general unifier σ of X and A. It then replaces langmember(σX,L,σV) by σC1 in σC. It continues making these substitutions and replacements until no further such occurrences of langmember(X,L,V) can be found. The result is D. Thus expandconditions(C,D,allsubs) provides a complete set of unifiers σ and conditions D such that D implies σC.

Expandconditions(C,D,always) also replaces all occurrences of the term langmember(X,L,V) where X is not a variable and L is not "all", but instead of using unification, it uses subsumption. It looks for rules of the form

languagerule(test,N,langmember(A,L,V),C1)

where there exists a substitution λ such that λA = X, and replaces X in C by λC1. It continues in this way until no further such occurrences of langmember(X,L,V) can be found. It then produces the disjunction D of all conditions E produced in this way.

Suppose now that we are attempting to prove that X implies Y. Computing all instances of expandconditions(X,Z,allsubs) will give a collection of pairs (σX,Z) which give a complete description of the conditions under which X is true. In other words, λX is true if and only if there is a (σX,Z) produced by expandconditions such that $\lambda = \mu\sigma$ for some μ and μZ is true. Our next job is to determine that, for each such pair, that Z implies σY. We thus need to compute the conditions under which σY is true. This is done using expandconditions(σY,U,always). Since U is the disjunction of a set of conditions under which σY is true, we have that U implies σY. We have thus reduced our problem to proving that Z implies U in each case. But this can be often be done using Boolean algebra, since many of the same conditions will appear in Z and U. Thus our last step is to use Boolean algebra to show that Z implies U.

As an example, consider the language defined by the following rules:

```
languagerule(test,1,langmember(rand(A,N),foo,V),ok).
languagerule(test,2,langmember(pke(pubkey(A),Y),foo,V),
   langmember(Y,foo,V)).
languagerule(test,3,langmember(pke(privkey(A),Y),foo,V),
   langmember(Y,foo,V)).
languagerule(test,4,langmember((X,Y),foo,V),langmember(Y,foo,V)).
```

Suppose that we want to show that membership of pke(X,Y) in foo implies that (X,Y), the concatenation of X and Y, is in foo. We first invoke

expandconditions(langmember(pke(X,Y),foo,V),Z,allsubs)

to obtain two solutions:

1. X = pubkey(A), Z = langmember(Y,foo,V), and;
2. X = privkey(A), Z = langmember(Y,foo,V).

Invoking

expandconditions(langmember((pubkey(A),Y),foo,V),U,always)

on the first solution yields U = langmember(Y,foo,V); the same value of U is given for the second solution. Thus, in each case it remains to be shown that langmember(Y,foo,V) implies langmember(Y,foo,V). This is trivially true.

The observation that Boolean algebra can help us allows us to define the following procedure in Prolog (somewhat simplified from the way it is implemented in the Protocol Analyzer):

```
impliesconditions(Langname,W1,W2) :-
    ifthen( expandconditions(langmember(W1,L,V),Expcond1,allsubs),
                ((    lookforconflicts(Expcond1)
        ;
expandconditions(langmember(W2,L,V),Expcond2,always),
                implies(Expcond1,Expcond2)
                ))
        ).
```

where the procedure ifthen(+A,+B) succeeds only if, whenever A succeeds, then so does B. For each expression Expcond1 and substitution σ produced by expandconditions, the procedure lookforconflicts(Expcond1) looks at Expcond1 and attempts to determine if E is never true (for example, if E is"A = B", where A and B cannot be unified). If that is the case, Expcond1 implies that σW2 is in L trivially. If lookforconflicts fails, the procedure

```
expandconditions(langmember(W2,L,V),Expcond2,always)
```

is used to produce an expression Expcond2 implying that σW2 is in L.

The procedure implies(+Expcond1,+Expcond2) is then used to prove that Expcond1 implies Expcond2. This is done by first putting each of Expcond1 and Expcond2 in normal form (A1; ... ;An) where Ai = (B1, ..., Bn) such that Bj is an atomic formula or of the form not(C) for some formula C, where C is also in normal form. We then attempt to show that, if Expcond1 = (X1; ... ; Xn) and Expcond2 = (Y1; ... ;Ym), then there exists a Yj that is implied by each Xi. Thus we are reduced to proving that (Z1,...,Zk) implies (W1,...,Wt). This is done for the most part by showing that each condition Wi either appears in (Z1,...,Zk) or is always true. There are a few cases, however, where it was worthwhile to institute special procedures. For example, one condition that comes up often is not(A = B), which is translated as not(caseof(A,B,V)), that is A is not subsumed by B. It is clear that not(caseof(A,B,V)) implies not(caseof(C,D,V)) if A is a subterm of C and B occurs in D the same place that A appears in C. This situation occurs often enough in our proofs that we have implemented it as a special case of the implies procedure.

We now define another recursive procedure, verifystates. The command

verifystates(+L,+Conds,+S)

succeeds if every path to the state S contains at least one word that can be proved to be a member of L using Conds. It begins by finding each state T preceding S,

and then uses impliesconditions to show that, there exists a word W known by the intruder in T such that Conds implies that W is a member of L. If that fails, it invokes verifystates(L,Conds,T). Verifystates fails if S is an initial state, that is, if S consists of words known by the intruder initially and lfacts that hold initially.

We are now ready prove that a language L is unreachable. We do this by invoking the procedure verifystates(L,langmember(W,L,V),[W]). In other words, we attempt to show that, if W is a member of L, then in any state preceding a state in which W is known by the intruder, some other member of Langname is known by the intruder. If verifystates succeeds, we will know that we succeeded in proving this.

We note that neither verifystates nor expandconditions is guaranteed to terminate. We guarantee termination by including limits on the number of times each procedure calls itself. These can be increased by the user. We also improve efficiency of verifystates by having findstate look for how to reach a subset of a state instead of the entire state; except in the first invocation, where it queries a single word, it only queries the substate of a state consisting of state variables, and ignores words known by the intruder. This makes for a much smaller search space, although it may mean that verifystates may fail in cases in which language unreachability is actually provable. In practice, though, this has happened seldom enough, and the improvement in efficiency has been great enough, that we feel justified in implementing findstate in this way.

3.4. How the User Interacts with the Analyzer

The user interacts with the Analyzer by first entering the description of a state. The Analyzer then responds with a description of all states that that can immediately precede that state, minus the states that are eliminated by the checks described above. Once this is done, there are two options that can be followed, depending upon whether the Analyzer is being operated in manual or automatic mode. If the Analyzer is being operated in manual mode, the user queries each state produced individually. The user has the option of only querying part of each state, that is, asking how some subset of the words and state variables that make up the state description is reachable, rather than the whole description. This option gives the user the ability to reduce the size of the search space without jeopardizing the soundness of the unreachability proof, since the unreachability of the substate implies the unreachability of the complete state, while the substate generally will have a smaller set of descriptions of preceding states than the complete set. However, if a subset is queried and is then shown to be reachable, it may be necessary to look at the entire state again to determine if it is reachable. The user has the option of "undoing" a portion of the search tree, as described in Section 3.1, and requerying it if this turns out to be the case.

It is also possible to use the Analyzer in automatic mode. In this mode the Analyzer queries each state itself after the first one, instead of leaving the job to the user. The Analyzer uses some simple heuristics for querying substates when used in automatic mode (for example, it will not ask how to find a word that the intruder is assumed to know initially), but in general it queries the whole state or close to it. Thus, although it is possible to generate states much faster when the Analyzer is used in automatic mode, in general the amount of states generated will be larger. It is possible, however, to switch back and forth between automatic and manual mode. Thus, if the search tree is becoming too bushy, the user can

switch back to manual mode, undo that part of the tree, query it manually, and then switch back to automatic mode. This may be done several times in a search.

The Analyzer is generally used in two phases. In the first phase, the user builds up the state conditions database and the language database. In the second phase, the user specifies insecure states and attempts to prove them unreachable. It is possible, and usually, necessary, to switch back and forth between the two stages. For example, the user may enter the second stage and then find that the state space generated is too large to be searched in a reasonable amount of time, or that an infinite loop that suggests a formal language is being generated. In that case he or she may go back to the first stage and attempt to build up the state conditions and language databases further.

An example of this search strategy in use is supplied in [9].

4. DEVELOPMENTAL HISTORY OF THE ANALYZER

The Protocol Analyzer has been developed in several phases. Each phase supplied a greater amount of automated assistance to the user. When a phase was complete, it was used to verify a number of protocols, and procedures that were repetitive and susceptible to automation were identified. These procedures were then automated and included as Analyzer functions, and the Analyzer was tested further.

To date, there have been three main phases. In the first, the Analyzer did little more than give a complete description of all states that could immediately precede a given state. This version of the Analyzer was used to verify several simple protocols. While the Analyzer was in this state, several general procedures for proving classes of states unreachable were developed. These included the use of formal languages described in the previous section.

In the next phase, the user was given the option of directing the Analyzer to avoid states that had been proved unreachable. Thus, if the Analyzer came up with an answer in which the state immediately preceding the specified state was one that had previously been proved to be unreachable, that answer would be rejected. At this point, we used the Analyzer to examine a number of open literature protocols. Although the Analyzer was still somewhat cumbersome to use, we were able to find previously undiscovered flaws in two such protocols: the Simmons selective broadcast protocol and the Burns-Mitchell resource sharing protocol discussed earlier in this paper. This convinced us that we were on the right track, and we proceeded to automate the Analyzer further.

In the present version of the Analyzer, it is possible, not only to record the results of hand proofs, but in many cases to perform the proof automatically. For example, as we described earlier, the Analyzer can be used to prove inductively that the intruder cannot learn any word of a specified formal language. At this point we also introduced the automatic search feature. This makes the search much easier to conduct when the search space has become small enough to search exhaustively. We have continued to apply the Analyzer to open literature protocols, both those that were known and those that were not known to be flawed. The Neuman-Stubblebine Aziz-Diffie protocols was analyzed when the Analyzer was in this state.

Our goal for the next phase is, not only to have the Analyzer be able to prove lemmas automatically, but to also have it give some assistance in generating lemmas to be proved. Thus, it may be possible, for example, to have the Analyzer generate candidates for formal languages that can be proved unreachable.

Finally, we note that the ability to prove unreachability of insecure states solves only half the problem. For cryptographic protocols, as in other areas, one must be able to determine what the desirable and undesirable properties are. To this end we have been developing a requirements language to be used with the Protocol Analyzer. In this language one specifies security requirements as requirements on sequences of events. The negation of the requirement can then be taken as an insecure state to be proven unreachable by the Protocol Analyzer. We have already applied this language to specify requirements for message authentication protocols [15], for key distribution protocols in which a session key is distributed by a trusted server [16], and for key agreement protocols, in which session keys are constructed out of secret information supplied by both parties [17], and we have found it to be useful both in providing goals for the Protocol Analyzer to verify, and bettering our understanding of the requirements of a protocol and the system it is to be used in.

5. THE USE OF PROLOG IN THE ANALYZER

The NRL Protocol Analyzer has been implemented in both Quintus Prolog and SWIProlog and at this point consists of about 4,000 lines of code. We chose Prolog as the implementation language for two reasons. The first was that, at the time this project started, very little was known about what techniques would be helpful in the analysis of cryptographic protocols, and what would not. Thus, when we tried an approach, we had no way of knowing beforehand whether or not it would be useful. Thus we needed a language that would support rapid prototyping and would allow us to try a number of different approaches in a short amount of time.

Another reason for our choice of Prolog was that the Protocol Analyzer is based on equational unification. Since Prolog is based on unification, this made it a natural choice for an analysis tool that uses narrowing as the basis of its state space search technique. This was especially helpful in languages like Quintus that offer unification with occur check.

Prolog has served us well in these respects. In particular, although we were initially interested in unification mainly as a means to support the implementation of narrowing algorithms, we found out as we progressed in the development of the Analyzer that we often had need to make use of a subtle interplay between some subset of identity checking, subsumption, unification, and narrowing. For example, use of the database of requirements on reachable states requires first subsumption, to determine whether or not a state contains an element of the database, and then unification, to guarantee that that element can be made to satisfy the necessary requirements. The interplay is even more subtle in the verification of language unreachability. In order to prove a language unreachable, we first run the Analyzer on each production W of the language to determine what sets of words X must be known before that word could be produced, thus making use of narrowing. We then determine under what conditions W can be proved to be a member of the language, using unification, and, for each such unification and set of conditions, we determine the conditions that guarantee that a member of X will belong to the language, using subsumption. Boolean algebra and checking for identity is then used to verify that the first set of conditions implies the second. Such delicate reasoning about the various variations on unification was not only relatively straightforward to implement in Prolog, but the Prolog paradigm, being based on unification, helped us to understand it more clearly.

We also found Prolog very useful as a rapid prototyping tool. We were able to produce a number of different versions of the Protocol Analyzer very rapidly, each one including new techniques and functions that we could test on examples and decide whether or not to keep. One case for which this was particularly helpful was in incorporation of procedures for verification of language unreachability. This is one of the more time-consuming operations in the Protocol Analyzer, and one of the ways we cut down on its expense is by reducing the sorts of situations in which the procedure will successfully verify a language to be unreachable. On the other hand, we do not want to reduce the scope of the cases handled so much that the procedure is not adequate to the types of situations that are likely to occur. The use of rapid prototyping made it easy to try out several different versions of the unreachability verification procedure and rate them according to speed and usefulness.

Rapid prototyping also made it possible to defer automation of procedures until we had used the Analyzer enough to identify procedures that were used often enough and were repetitive enough so that automation was both possible and useful. This allowed us to maximize the benefit we got from increasing the automation of the Analyzer. For example, the language unreachability verification procedure was derived from examining the way in which such verification was done manually on output produced by the Analyzer.

We also found a few drawbacks, although in many cases it was hard to determine whether these were to blame on Prolog itself or the way in which we used it. Most notable of these is the way in which the Protocol Analyzer relies on a "generate and test" strategy. A set of candidate states preceding a goal state is generated, and then a number of tests are run to eliminate unreachable states from the set. This had two advantages for us: one, since Prolog operates by producing all solutions to a query when it is questioned repeatedly, it is straightforward to implement such a generate and test strategy. It is also an easy strategy to adapt to rapid prototyping. Whenever we found a new way to prove that a class of states was unreachable, we simply implemented it as a new kind of test. However, the generate and test strategy can also be highly inefficient. In particular, towards the end of an analysis we usually find that the the Protocol Analyzer will be flunking considerably more candidate states than it passes, and thus as we progress in an analysis we find a marked reduction in the apparent rate at which the Protocol Analyzer produces new states. It is possible that a language that supported a more efficient search strategy would result in a more efficient implementation of the Protocol Analyzer.

Other problems we had were more mundane. Although we found the degree to which Prolog supported the extensive interplay of subsumption, unification, and narrowing very useful in the implementation of the Analyzer, it made it harder to explain the Analyzer to those who were not familiar with Prolog and unification. We also found the lack of standardization of Prolog a problem when we tried to port the Analyzer from one version to another; this was in particular a problem when we used built-in functions.

6. CONCLUSION

In this paper we gave an overview of the NRL Protocol Analyzer, an interactive software tool for the analysis of cryptographic protocols. We showed how the Analyzer works and how it is used, and we described its achievements so far. We

also described the ways in which our choice of the Prolog language influenced the development of the Analyzer. Although the Analyzer is still under development, it has had a number of significant successes. As we continue to develop and refine it, we can hope for many more.

REFERENCES

1. A. Aziz and W. Diffie. Privacy and Authentication for Wireless Local Area Networks. *IEEE Personal Communications*, 1(1):25–31, 1994.
2. J. Burns and C.J. Mitchell. A Security Scheme for Resource Shoring Over a Network. *Computers and Security*, 9:67–76, February 1990.
3. Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. *ACM Trans. on Computer Systems*, 8(1):18–36, February 1990.
4. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.
5. W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions in Information Theory*, IT-22:644–654, 1976.
6. Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes, and Cryptography*, 2:107–125, 1992.
7. D. Dolev, S. Even, and R. Karp. On the Security of Ping-Pong Protocols. *Information and Control*, pages 57–68, 1982.
8. D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.
9. Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three Systems for Cryptographic Protocol Analysis. *Journal of Cryptology*, 7(2), 1994.
10. C. Meadows. A System for the Specification and Analysis of Key Management Protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195. IEEE Computer Society Press, Los Alamitos, California, 1991.
11. C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1:5–53, 1992.
12. Catherine Meadows. Formal verification of cryptographic protocols: A survey. In *Proceedings of AsiaCrypt '94*, 1994. to appear.
13. B. Clifford Neuman and Stuart G. Stubblebine. A Note on the Use of Timestamps as Nonces. *Operating Systems Review*, 27(2):10–14, April 1993.
14. G. J. Simmons. How to (Selectively) Broadcast a Secret. In *Proceedings of the 1985 IEEE Computer Society Symposium on Security and Privacy*, pages 108–113. IEEE Computer Society Press, 1985.
15. Paul Syverson and Catherine Meadows. A Logical Language for Specifying Cryptographic Protocol Requirements. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE Computer Society Press, Los Alamitos, California, 1993.
16. Paul Syverson and Catherine Meadows. Formal requirements for key distribution protocols. In *Proceedings of Eurocrypt 94*. Springer-Verlag Lecture Notes in Computer Science, to appear 1994.
17. Paul Syverson and Catherine Meadows. A formal language for cryptographic protocol requirements. submitted for publication, 1995.