The NRL Protocol Analyzer: An Overview

Catherine A. Meadows
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington DC, 20375

Abstract

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 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 [4], 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 thown 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 [5], the following attack is presented:

Let I be the intruder.

1. $I \hookrightarrow B$: 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. $I \hookrightarrow A$: 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 [2, 3, 9, 10, 12]. 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 [2]. Another is to use some form of model checking, in which one models the protocol as an interaction between a set of state machines and attempts to locate security flaws by working backwards from an insecure state. This is the approach taken by the NRL Protocol Analyzer.

A number of challenges exist that must be met when applying model checking 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, the it we encrypt a word over and over with a key, we assume that we produce an unbounded set of words.

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 [12] and the Burns-Mitchell Resource Sharing Protocol [1], and a previously unknown implementation-dependent flaw in the Neuman-Stubblebine reauthentication protocol [11]. The results of these analyses are contained in [9, 10, 13].

2 Description of the Analyzer

In this section we describe how the Analyzer works, and outline the basic features used for analysis of cryptographic protocols. A more complete presentation, with a worked example, may be found in [8].

The Analyzer is based upon a version of the term-rewriting model of Dolev and Yao [7]. 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 [6], 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, 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.

Transition rules can also describe the actions of an intruder who produces new messages out old by performing some operation such as encryption or decryption.

Transition rules generally involve variables. For example, a transition rule that describes the action of a principal initiating a protocol will have a variable standing for the principal name, so the rule holds for all principals. Likewise, if a rule requires that a principal receive a message, this message can be represented by a variable, to reflect that fact that the intruder can substitute any word for that message. Following the Prolog convention, variables are represented by words beginning with capital letters and quantification is assumed to be universal.

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.

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. Both words known by the intruder and values of local state variables 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.

Once this is done, the user may query each of the preceding states in turn. He or she can then query the states immediately preceding those states, and so on.

If such a search is performed without any further control by the user, it will never end, since the search space is infinite. Thus the Analyzer includes a number of means by which the search can be controlled by the user. We describe these below.

2.1 Partial Queries

When the Analyzer presents an input state, the user has the option of querying a portion of the state instead of the whole state. For example, if the state consists of a word known by the intruder and the assertion that a state variable has a certain value, the user can ask the Analyzer how that state variable can be reached without asking how the word can be reached. This makes the amount of information the Analyzer has to match smaller, and thus reduces the search space. Moreover, proofs of the unreachability of insecure states are still valid, since, if a piece of the state description is unreachable, then so is the entire state. However, a partial state may be reachable while the entire state is not. Thus, if an attack (that is, a path to an insecure state) is found by this method, it must be reverified by queries on the complete states to determine whether or not it is a valid one.

The Analyzer can be used in an automatic mode in which each state produced is queried automatically by the Analyzer using simple heuristics for making partial queries. We will describe this function in more detail later on.

2.2 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.

2.3 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:

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

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

We want to show that it is impossible to find any word of **F** unless some other irreducible words of **F** (including e(k,X)), have already been found. 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.

We have developed an algorithm for checking language unreachability that makes use of the technique we described above, and we have incorporated that algorithm in the Protocol Analyzer. The algorithm is pessimistic; that is, it fails to prove that some unreachable languages are unreachable, but if it determines that a language is unreachable, then it is in fact unreachable.

2.4 Automatic Mode

It is possible to use the Analyzer in automatic mode, in which the user defines a goal state and the Analyzer searches for a path to it in a breadth-first fashion. When used in automatic mode, the Analyzer applies some simple heuristics for tree pruning via partial queries: when it is trying to determine how to find a state it does not look for empty state variables, words known by the intruder initially, or words represented by variables. It is possible to switch in and out of automatic mode, and usually this is necessary. At some points the search tree will become too bushy, and it will be necessary to prune it either by performing partial queries manually, or by generating new lemmas on unreachable languages and conditions for state reachability. In some of the simpler protocols though, we have been able to reproduce attacks by first proving a few straightforward lemmas about language unreachability and state reachability conditions and then running the program in automatic mode. For example, we were able to reproduce the attack on the protocol described in Section 1 this way.

3 Developmental History and Achievements of 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 it 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. 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 particularly helpful was in incorporation of algorithms 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 algorithm 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 algorithm is not adequate 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 algorithm 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 original language unreachability verification algorithm was derived from examining the way in which such verification was done manually on output produced by 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.

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 proving 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.

4 Conclusion

In this paper we gave a brief 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.

References

[1] J. Burns and C.J. Mitchell. A Security Scheme for Resource Sharing Over a Network. *Computers and Security*, 9:67–76, February 1990.

[2] Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. *ACM Trans. on Computer Systems*, 8(1):18–36, February 1990.

[3] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[4] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions in Information Theory*, IT-22:644–654, 1976.

[5] W. Diffie, P. C. Van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[6] D. Dolev, S. Even, and R. Karp. On the Security of Ping-Pong Protocols. *Information and Control*, pages 57–68, 1982.

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

[8] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *The Journal of Cryptology*, to appear 1994.

[9] 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.

[10] C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1:5–53, 1992.

[11] 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.

[12] 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.

[13] Paul Syverson and Catherine Meadows. Formal requirements for key distribution protocols. submitted for publication 1994.