J. Cryptology (1994) 7: 79–130

Journal of
CRYPTOLOGY

Three Systems for Cryptographic Protocol Analysis

R. Kemmerer
Computer Science Department, University of California,
Santa Barbara, CA 93106, U.S.A.

C. Meadows
U.S. Naval Research Laboratory, 4555 Overlook Avenue, SW,
Washington, DC 20375, U.S.A.

J. Millen
The MITRE Corporation, Bedford, MA 01730, U.S.A.

Communicated by Thomas Beth

Received 25 March 1992 and revised 17 August 1993

Abstract. Three experimental methods have been developed to help apply formal methods to the security verification of cryptographic protocols of the sort used for key distribution and authentication. Two of these methods are based on Prolog programs, and one is based on a general-purpose specification and verification system. All three combine algebraic with state-transition approaches. For purposes of comparison, they were used to analyze the same example protocol with a known flaw.

Key words. Interrogator, Narrower, Inatest, Ina Jo, Key distribution, Authentication, Security, Protocols, Formal methods, Specification, Verification.

1. Introduction

1.1. *Background*

Formal methods, including both specification and verification, are used in computer science to verify the correctness of systems that are too complicated and subtle to be easily understood, and whose correct operation is vital enough so that a high degree of assurance is desired. First, the system is specified in a formal specification language that has some mathematical basis; then theorems are proved about the specification with the assistance of an automatic theorem-prover. Something is usually learned about the system, not only from the results of attempting to prove it correct, but from the greater understanding that comes from expressing it in a formal specification language.

Techniques for the specification and verification of both the security of computer

systems and the correctness of communication protocols have been in existence for some time. See Holzmann's book, for example, for a survey of communication protocol verification [11]. As Kemmerer [12] has pointed out, it seems natural that such techniques should be applied to cryptographic protocols.

Cryptographic protocols are relatively simple in structure, but their security properties are not always intuitively obvious, and the number of protocols that have appeared in the literature that subsequently were proved to be flawed suggests that more formal analysis is necessary. The peculiar difficulty of cryptographic protocol security analysis arises from the threat of malicious interference, where an attacker may intercept and replace or modify messages in transit. While this threat applies to noncryptographic protocols as well, cryptographic protocols are, or should be, designed to foil such attacks, and it is difficult to design them properly and to prove that the design is effective.

A great deal of work has been done in applying mathematical techniques both to cryptography and to developing security models for protocols that are directly based on "hard" problems (e.g., in [10]), but until recently very little attention had been paid to applying machine-assisted formal verification techniques to them.

In the last few years, however, the situation has changed. A number of researchers have begun applying formal verification techniques to cryptographic protocols, in many cases with very interesting results. For example, Burrows *et al.* have developed a logic for analyzing authentication protocols and show how the use of the logic uncovered previously unknown flaws [5]. Similarly, in [18], Meadows used one of the systems described in this paper to find a previously unknown flaw in Simmons's selective broadcast protocol [26], and in [17] she showed how the effort of capturing the security goals of a protocol in a formal specification language led to the discovery of a previously unknown flaw in the Burns–Mitchell resource-sharing protocol [4]. In this paper we give an introduction to this new field by providing an in-depth view of three experimental systems that have been used to determine how useful formal methods can be in the verification and understanding of cryptographic protocols. Two of these systems, the NRL Protocol Analyzer and the Interrogator, were developed expressly for the analysis of cryptographic protocols; the third, Inatest, is a specification execution tool designed to support general-purpose software specification and verification using the Formal Development Methodology. In order to set off the similarities and differences between them better, all three systems were applied to the analysis of the same protocol, which was already known to have a security flaw. The specifications and results of the analyses are described in this paper.

Generally, some combination of three approaches is taken in the application of formal methods to the verification of cryptographic protocols. The first approach, initially formulated by Dolev and Yao [8], we call the *algebraic* approach. With this approach, a protocol is modeled with a collection of rules for transforming and reducing algebraic expressions representing messages. An example of a rule is the fact that encryption and decryption cancel each other out. Rules are invoked as a result of actions by communicating parties or an intruder. An analysis is performed by determining whether or not there is any sequence of rule applications that results in a subversion of the protocol's goals. Longley implemented rules of this type with an expert system [15].

The second approach is the *state-transition* approach, originating in the formal analysis of protocols for functional concerns such as deadlock and undefined transitions. This approach tests whether insecure states are reachable. State transition rules can be implemented within an algebraic system, as Meadows has done. The third approach is called the *logical* approach: a set of axioms is developed describing the conditions under which certain knowledge and/or beliefs will hold, and the protocol is analyzed according to that logic. The best-known example of such a system is that of Burrows *et al.* [5]. While the analysis techniques used in a logical system are rather different from those using state-transition models, it is interesting to note that the logical approach can be given state-transition semantics [1], and that logics have been combined with algebraic models by Merritt and Wolper [19] and the work that builds on theirs [2], [29].

Logical systems arising from [5] consititute an important and fruitful line of research for protocol analysis. We, however, confine this paper to a comparison of three systems that combine algebraic with state-transition approaches. By doing so, we avoid a substantial set of complex issues surrounding features such as the "idealization" process that turns a protocol into a set of logical assertions, that are peculiar to the logical approach.

1.2. *The Three Systems*

While the three systems described in this paper all combine algebraic with state-transition approaches, each system implements them in a different way.

In Kemmerer's system the protocol and the means by which an intruder attacks it are described in a formal specification language, Ina Jo, which had been designed as a general-purpose tool to support software development and correctness proofs. A symbolic execution tool, Inatest, is then used to "walk through" the protocol and demonstrate its vulnerabilities. Kemmerer has also used an Ina Jo approach on other protocols to identify flaws, and he has used formal verification techniques to prove security properties of these protocols [12].

Millen's and Meadows's systems search backward instead of forward. In these systems the user first specifies an insecure state, and the system is used to determine whether or not that state is reachable. In Millen's Interrogator the protocol is specified symbolically in Prolog, using predicates to describe the state transitions. Its handling of encryption is built in. The Interrogator performs an exhaustive backward search through a single run of the protocol to determine if the insecure state is reachable from an initial state. The user can control the extent of the search by varying the assumptions about what information is available to the intruder. For example, the user might select certain information from a normal run of the protocol that he thinks would be useful.

In Meadows's tool, the Naval Research Laboratory (NRL) Protocol Analyzer, the search is less automated, and unlike the other two systems, its main intent is to assist the user in proving a protocol secure rather than to uncover vulnerabilities. The protocol is specified in Prolog as a set of state transition rules describing the sending and receiving of messages. The words used in the messages obey a set of reduction rules. In the Protocol Analysis Tool the user specifies an insecure state, and the tool gives the user a complete description of all states that can immediately

precede that state. The user can then query these states to find out from what states they are reachable, and so forth. The user can limit the search space by proving that portions of it are unreachable; for example, the tool assists the user in detecting and avoiding infinite loops.

It is also possible to use the NRL Protocol Analyzer in "automatic" mode, so that a search tree is produced without any input from the user except the provision of a final goal. This should not be done, however, until the user has limited the size of the search space by proving lemmas about unreachability of portions of it. Even then, it may require intervention from the user from time to time to reduce the size of the tree, by the use of such techniques as querying subdescriptions of a state instead of the entire state.

Each of these systems offers its unique advantages. With Inatest, the use of forward search means that a protocol can be explored to discover insecure states that may not have been envisioned by the protocol designer. The Interrogator provides the greatest amount of assistance in determining flaws for the least amount of effort; it is merely necessary to specify the protocol and make assumptions about what information available to the intruder is relevant. The NRL Protocol Analyzer is the most useful in providing a high degree of assurance that a protocol is secure; however, it has also been used to uncover subtle security flaws.

In this paper we restrict ourselves to the analysis of a key distribution protocol. However, we note that the systems discussed in this paper can be and have been applied to other kinds of protocols. In particular, the NRL Protocol Analyzer has been applied to protocols designed for selective broadcast of messages [18] and the secure sharing of resources [17], and Kemmerer has applied Ina Jo to a key management facility [12].

1.3. *The Example Protocol*

In this paper we demonstrate and compare how the three systems work by showing how they were used to analyze a single flawed protocol, due to Tatebayashi *et al.* [28]. The Tatebayeshi–Matsuzaki–Newman (TMN) protocol is a key distribution scheme by which a pair of ground stations in a mobile communication system obtain a common session key, through the mediation of a trusted server. Messages to the server are encrypted in the server's public key, known to all ground stations. Ground stations generate conventional keys to be used as session keys. The protocol was found by Simmons to suffer from several security flaws, which are documented in [28]. Simmons suggested this protocol as a test case for the different protocol analysis systems.

Success on this example does not prove the power of the three systems, since Simmons's analysis was known ahead of time, and this particular example does not bring out some of the special strengths available in these systems. The example is simply a vehicle for illuminating such issues as specification style, user interaction, and differences with respect to the kind of informal analysis done by an expert such as Simmons.

The TMN protocol works as follows. The server possesses a public–private key pair. The public key is known to everyone in the system, while the private key belongs to the server alone.

1. When user A wishes to communicate with user B, it encrypts a random number with the server's public key, and sends the encrypted random number, along with A's and B's names.
2. When the server receives the request, it decrypts the random number and stores it as a key-encryption key for that conversation; it also notifies B that A wishes to speak to it.
3. User B, on receiving the notification from the server, generates a random number to be used as a session key, encrypts it with the server's public key, and sends it to the server.
4. The server decrypts the response, encrypts the key with A's random number using a private-key algorithm, and sends the result to A. In the specification of the protocol, the algorithm used for this step is commutative in the sense that, if $A[B]$ represents the encryption of B under the key A, we have $A[B] = B[A]$. This could be done with a bitwise exclusive-or of the two keys, for example. The commutativity is significant for Simmons's analysis and Kemmerer's demonstration.
5. User A decrypts the message from the server using the original random number it had generated and assumes that the result is the session key.

The protocol is diagrammed in Fig. 1. In this diagram the server's public key is e, user A's key is $r1$ and user B's key is $r2$. Encryption is shown generically in the form $key[data]$.

This protocol suffers from at least two security flaws. One results from the fact that the public-key encryption algorithm used is a homomorphism, and thus can be used by an intruder to construct words that the protocol designer had not intended. This flaw, discovered by Simmons, is discussed in detail in [28], so we omit its description here. The other flaw results from the fact that no secure authentication is used between parties, and so an intruder who is able to intercept messages can, by impersonating user B to the server, cause it to convince A that a key generated by the intruder is a key generated by B. In this paper we show how Inatest can be used to uncover the first flaw, while the NRL Protocol Analyzer and

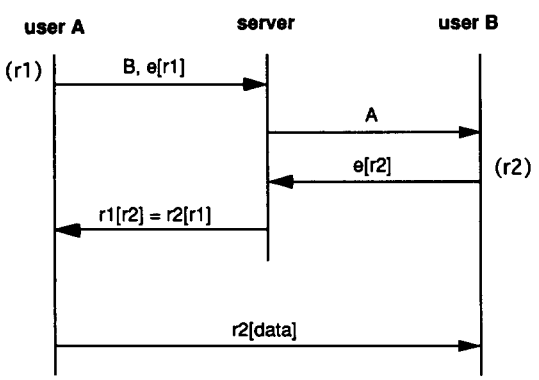

Fig. 1. The TMN Example Protocol.

Interrogator can be used to find the second. We present the Interrogator first, then the NRL Protocol Analyzer, and finally the use of Ina Jo and Inatest.

2. The Interrogator

2.1. *Introduction*

The Interrogator is a Prolog program that searches for active wiretapping vulnerabilities in protocols. It is automatic in the sense that once the protocol and initial assumptions have been specified, the program runs by itself until it either finds a penetration scenario or gives up.

The earliest form of the Interrogator was documented in [20]. A more capable and flexible version for the Symbolics LISP machine, with graphic display of protocols, was described in [21]. This version of the program was able to reproduce the Denning–Sacco flaw [7] in the Needham–Schroeder protocol [23]. There are several examples of protocol flaws which it can find, though to date it has not yet been responsible for discovering any previously unknown vulnerability. Presently, the Interrogator runs on a Macintosh, and there is a new experimental version of it under development. Most of the present description applies to a relatively stable "baseline" version of the Interrogator as of April 1990, which is similar in capability to the 1987 version.

Protocol specifications for the Interrogator model a protocol as a set of communicating state-transition machines, each of which represents a party. The state of each party includes a list of memory items that expands as it receives data from messages. A state transition occurs in a party when a message is sent or received. Because we assume there is an attacker capable of message modification, a message received is not necessarily the same as the message sent in the previous (or any earlier) transition.

The general idea of the Interrogator is to define a predicate, pKnows(D, H, B, S), which holds when the penetrator is able to obtain knowledge of specified data D, via a message history H, which takes the network from its initial state to an insecure goal state specified by B and S. By invoking pKnows with H as a variable, the user causes the Prolog interpreter to search for an acceptable instantiation for it. The resulting message history H is the penetration scenario.

The Interrogator's search strategy is dictated largely by the normal operation of the Prolog interpreter. Like the NRL Protocol Analyzer, it begins from the goal state and works backward, finding the possible prior states according to the protocol specification. Unlike the NRL program, which (except in automatic mode) presents all the possible prior states to the user in breadth-first fashion, and asks for the user's selection to continue the search path, the Interrogator continues automatically in depth-first fashion to look at earlier states. Each branch of the search path is terminated either by a success when it reaches the initial state, or by a dead end when it reaches a state that has no possible prior state.

2.2. *Program Structure*

The program is conceptually divided into two fixed sections, the Knowledge and Reachability sections, plus two replaceable sections, the Protocol and Scenario

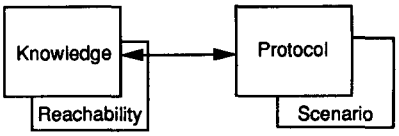

Fig. 2. Conceptual structure of the Interrogator.

sections, illustrated in Fig. 2. The Knowledge section contains the predicate pKnows, and gives rules by which the penetrator extracts data from a message. If a message contains encrypted data, the penetrator obtains it only if the penetrator knows the appropriate key. Thus, the Knowledge section may call itself recursively.

The Reachability section determines whether a legal message history taking the network from its initial state to a given state exists. It attempts to backtrack over one state transition, and then calls itself recursively. To identify legal state transitions, it calls the Protocol section. The Reachability section also defines the possible ways a penetrator might upset the normal course of a protocol by intercepting and modifying messages.

The Protocol section is essentially the protocol specification. This section is constructed specially for each protocol to be analyzed. A protocol is modeled as a collection of communicating parties, each of which is a state-transition machine. The network state is the combination of the current states of all the parties, plus a "network buffer" which may hold a message in transit. A state transition for a party is either a "transmit" or "receive" transition, specified in Prolog as a relation among a prior state, a message, and a next state. The state of a party has some structure to be described later.

The Scenario section specifies assumptions that help to define the possible penetration scenarios. Here, the analyst states which items of data are assumed known to the penetrator initially. The analyst may also have identified some constraints or suspicions about the final state after the penetration. The more information that is supplied to constrain the final state, the fewer search paths the Interrogator must consider, and the faster it runs.

While a state-machine representation of protocols is common to the other analysis approaches described in this paper, the Interrogator differs by not having the set of data items known to the penetrator saved as part of the current network state.

There is a close relationship between the Knowledge and Reachability sections.

In order for the penetrator to obtain data, that data must have been transmitted in a message by some party. That happens only when the party is in a certain state, as specified in the protocol. The program then asks whether that state is reachable. This is important, because, if the state is not reachable, then, despite appearances, it is really not possible for the penetrator to have obtained the data that way.

On the other hand, in order for a state to be reachable, it may be necessary for the penetrator to modify a message, inserting some data known to the penetrator. This leads to a call on pKnows, to find out whether the penetrator could have obtained that data.

2.3. *The Encryption Model*

The baseline version of the Interrogator has a very simple model of encryption. Encrypted data in messages is represented symbolically by a list [key, data]. Built into the Knowledge section is the simple fact: If the penetrator knows the key and the encrypted field [key, data], then it knows the data, at least in a single-key system.

Public-key encryption is handled by defining a relation inverseKey(pkey, skey). If the penetrator knows pkey and [skey, data], it has found data; or if it knows skey and [pkey, data], it has found data. The two keys are treated symmetrically; the only distinction between public and secret keys is the initial assumption that the penetrator knows the public one. A conventional key (that is, a key used in a single-key encryption system) is its own inverse.

In the more recent experimental version of the Interrogator, a fifth section has appeared: the Algebra section. This contains code for simplifying expressions containing symbolic computations occurring in messages. These expressions indicate encryption and other operations explicitly. Different encryption operators distinguish public-key from symmetric-key encryption, encryption from decryption, and so on. For example, dec(k, enc(k, x)) is simplified to x, and xor(x, xor(y, x)) is simplified to y, where "xor" represents bitwise exclusive-or, or modulo-two addition.

2.4. *Penetrator Actions*

Recall that the network state consists of the states of all parties, plus the contents of the network buffer. A transmit transition causes a message to be placed in the network buffer, and a receive transition causes the message currently in the network buffer to be removed. The penetrator action, if any, occurs between a transmission and the next reception. As an active wiretapper, the penetrator is assumed to be able to observe and manipulate the network buffer, as shown in Fig. 3. The penetrator can modify the message currently in the network buffer, by replacing any or all of the fields in the message. Data items inserted as replacements must be known to the penetrator.

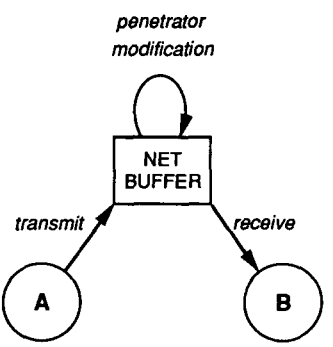

Fig. 3. Message transmission, modification, and reception.

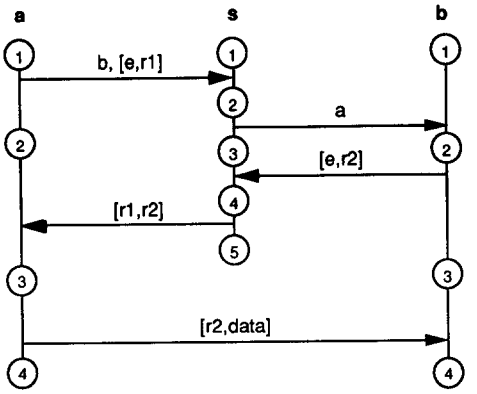

Fig. 4. A Normal Run of the TMN Protocol.

2.5. *The Protocol Example*

A normal run of the TMN protocol is illustrated schematically in Fig. 4. This figure differs from Fig. 1 in that it shows state numbers, and it also uses the Interrogator's notation for encrypted expressions. The initiating party, a, sends a message to the server, s, indicating the desired receiving party, b, and supplying a new key $r1$ encrypted in the server's public key e. The server tells b that a wants to open a connection, and b responds with another new key $r2$, encrypted in the server's public key. The server forwards $r2$ to a, encrypted using $r1$. Now a and b share the key $r2$, which can be used as a session key. One use of $r2$ to encrypt some data from a is shown.

Note that the details of the encryption techniques are not visible in this representation. In particular, the encryption of $r2$ using $r1$ in the penultimate message in Fig. 4 can be handled in various ways. The original protocol suggested combining $r1$ and $r2$ arithmetically, with a sum, product, or quotient, which led to difficulties noted by Simmons in his analysis. The Interrogator can only make use of the abstract version, which tells it only that a, knowing $r1$, is able to extract and use $r2$. Simmons's analysis makes use of the additional property that a party knowing $r2$ can extract and use $r1$ as well. The current experimental version of the Interrogator is able to represent such arithmetic operations.

2.6. *Encoding the Protocol*

The main part of the protocol specification is a set of transmit and receive clauses showing how each party changes state as a result of sending and receiving messages. The state changes are indicated in Fig. 4, showing the succession of numerical state labels for each party. The state of a party includes not just the state label, but also a list of data items that the party has learned from previous messages and which are needed to construct later messages. These lists will appear in the transmit and receive clauses.

Consider the first message from a to s. Only the data fields in the message are shown on the arrow in Fig. 4. The full message is $[a, s, 1, b, [e, r1]]$. The first three items may be thought of as the "header." By convention, they are the source address, destination address, and a format label. As we shall see below, each message has a format specifying the data types for each of its fields. The format label helps the receiver of the message to recognize it and respond appropriately, and it also helps the Interrogator to limit its search for modified message field values to appropriate possibilities.

The state change for the first transmit operation is specified with this transmit clause:

$$\text{transmit}([a, 1], [a, s, 1, b, [e, r1]], [a, 2]).$$

The three arguments of the transmit term are

prior state, message transmitted, next state.

In general, a party state is encoded as a list of the form [party, state-label, ...] where the third and subsequent items, if any, are individual data items remembered by that party. In the transmit term above, we see that party a has not yet learned anything. Note that the transmit clause gives the state change for only one party, the one transmitting the message. This is sufficient because only one party changes state at a time.

It is, perhaps, confusing that the Prolog list data structure is being used to represent three distinct expression types: a message, an encrypted data field, and a party state. The role of a given list is determined by context, as is the significance of the items in each list.

The next state change that would occur normally is that of the server upon receiving the first message. This one is more interesting, since it adds some new information to the server's memory:

$$\text{receive}([s, 1], [A, s, 2, B, [e, R1]], [s, 2, A, B, R1]).$$

Note that the message fields that went into the server's memory are indicated with capital letters, which, by Prolog convention, are variables. The rationale for this is that the server does not know in advance who will request service, to whom a connection is requested, or what key will be specified. The server does assume that the key $R1$ is encrypted in the server's public key. In effect, $R1$ is defined to be the result of decrypting the last message field using the server's secret key. This clause defines a family of transitions for all possible values of the variables A, B, and $R1$.

To see how the server uses its memory, look at the next transmit clause:

$$\text{transmit}([s, 2, A, B, R1], [s, B, 3, A], [s, 2, A, B, R1]).$$

The server simply tells B (whoever that may be) that A wants a connection, and retains its memory of data items.

The complete protocol specification is shown in Fig. 5. Besides transmit and

```
------------------------------------------------------------
transmit([a,1],          [a,s,1,b,[e,r1]], [a,2]).
receive( [s,1],          [A,s,1,B,[e,R1]], [s,2,A,B,R1]).
transmit([s,2,A,B,R1],   [s,B,2,A],        [s,3,A,B,R1]).
receive( [b,1],          [s,b,2,A],        [b,2,A]).
transmit([b,2,A],        [b,s,3,[e,r2]],   [b,3,A]).
receive( [s,3,A,B,R1],   [B,s,3,[e,R2]],   [s,4,A,B,R1,R2]).
transmit([s,4,A,B,R1,R2],[s,A,3,[R1,R2]],  [s,5]).
receive( [a,2],          [s,a,3,[r1,R2]],  [a,3,R2]).
transmit([a,3,R2],       [a,b,4,[R2,data]],[a,4,R2]).
receive( [a,4,R2],       [b,a,4,[R2,_]],   [a,5]).

initialNodelist([[a,1],[s,1],[b,1]]).

pKnowsInitially(e).
pKnowsInitially(r3).
pKnowsInitially(X) :- ptype(X,addr).
pKnowsInitially(X) :- ptype(X,form).

mformat([_s,_d,1 | _a],[addr,addr,form,addr,[cryp,key]]).
mformat([_s,_d,2 | _f],[addr,addr,form,addr]).
mformat([_s,_d,3 | _e],[addr,addr,form,[cryp,key]]).
mformat([_s,_d,4 | _e],[addr,addr,form,[cryp,data]]).

ptype(a,addr).
ptype(s,addr).
ptype(b,addr).
ptype(x,addr).
ptype(e,key).
ptype(se,key).
ptype(r1,key).
ptype(r2,key).
ptype(r3,key).
ptype(1,form).
ptype(2,form).
ptype(3,form).
ptype(4,form).
ptype(data,data).

inverseKey(se,e).
inverseKey(e,se).
inverseKey(r1,r1).
inverseKey(r2,r2).
inverseKey(r3,r3).

hopeless(r1).
hopeless(r2).
hopeless(se).

state([[a,4,r3],[s,5],[b,1]]).
------------------------------------------------------------
```

Fig. 5. TMN protocol specification for the Interrogator.

receive clauses, it contains the format definitions and associated type declarations, the key relationships, the initial state ("initialNodelist"), and some scenario information, to be discussed below.

2.7. *Scenario Specification*

In order to run the Interrogator, a penetration goal must be identified, along with any assumptions about what information is already known to the penetrator.

For the TMN protocol, we assume that the penetrator is another ground station knowing the server's public key, and having the ability to generate potential session keys. This initial knowledge is specified with pKnowsInitially clauses. Besides the keys, the penetrator is also assumed to know public constants such as the addresses of other parties and the protocol formats.

The key that will be generated by the penetrator is represented by the symbolic constant $r3$, declared as initial knowledge. Although such keys are thought of as being generated on the fly when needed, it could just as easily be imagined that a private list of keys has been prepared ahead of time, and $r3$ is the next one to be used.

The goal item is the data transmitted by a (and intended for b). When invoking the Interrogator, we must indicate not only the goal item but also the final network state. While this could be left as a variable, it can save considerable search time to give the program some hints. In Fig. 5 the state clause specifies the goal state for each party: [[a, 4, $r3$], [s, 5], [b, 1]]. For party a, this says that a has just transmitted the data encrypted with the penetrator's key $r3$. This assumption embodies a particular hypothesis as to how the data might be compromised. Party s is in its final state. Party b is still in state 1. Again, this represents a hypothesis about the nature of the penetration; it says that all messages to party b will be intercepted.

Three keys are designated as "hopeless": $r1$, $r2$, and se. This saves the Interrogator the time it might otherwise spend trying to figure out if they can be compromised.

2.8. *Running the Interrogator*

The Interrogator is invoked with the goal state as shown in the first line of Fig. 6. The goal state, S, is the one satisfying state(S) as specified in the last line of Fig. 5.

After about 12 seconds on a Macintosh IIcx, the program prints "Found data!" and displays the penetration history H twice, once under program control and once as part of the normal Prolog output.

The message history shows the sequence of messages sent and received. Each message is received as it was sent unless the penetrator modifies it. One message, from s to b, was modified; it was turned into a response apparently from b to s, containing the penetrator's key instead of a key generated by b. A schematic picture of what happened is shown in Fig. 7.

What the penetrator was supposed to have done was to intercept the message from s, so that b does not receive it, construct the simulated reply, and send that reply directly to s. The reply was constructed out of information known to the penetrator at that point. In particular, since $r3$ is the penetrator's key, and e is a public key, it could encrypt the former with the latter to get the field [e, $r3$].

```
?- state(S),pKnows(data,H,[],S).
...
Found data!
[sent,[a,s,1,b,[e,r1]]]
[rcvd,[a,s,1,b,[e,r1]]]
[sent,[s,b,2,a]]
[rcvd,[b,s,3,[e,r3]]]
[sent,[s,a,3,[r1,r3]]]
[rcvd,[s,a,3,[r1,r3]]]
[sent,[a,b,4,[r3,data]]]
    S = [[a,4,r3],[s,5],[b,1]],
    H = [[sent,[a,b,4,[r3,data]]],[rcvd,[s,a,3,[r1,r3]]],
         [sent,[s,a,3,[r1,r3]]],[rcvd,[b,s,3,[e,r3]]],
         [sent,[s,b,2,a]],[rcvd,[a,s,1,b,[e,r1]]],
         [sent,[a,s,1,b,[e,r1]]]]
```

Fig. 6. Interrogator run on TMN.

In this penetration history the penetrator must pretend to be *b*. This is inferior to Simmons's penetration, in which the penetrator could eavesdrop on a complete conversation between *a* and *b* without making the effort to imitate *b*. Nevertheless, it successfully captures at least one data message intended for *b*.

2.9. *Sensitivity*

The principal drawback of the baseline version of the Interrogator is that the time it takes to find a penetration can vary wildly depending on details of the protocol specification and the assumptions made for the scenario. There are also (rare) situations where the program cuts off a search path that leads to a penetration, and so fails to find one, due to the limitations of certain heuristics that were built into the program to help speed up the search.

The program takes more time if less information is specified about the goal state. For example, if either the key used by *a* or the final state of *b* is left unspecified, the

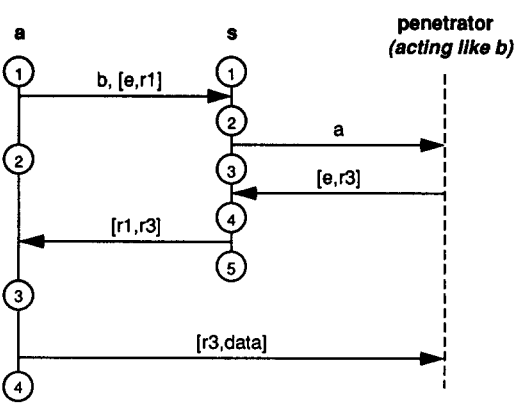

Fig. 7. The Interrogator's TMN penetration.

program runs for so long it has never seemed worth the time to find out exactly how long; at least 10 minutes. If the final state of *b* is specified as state 3, the program finds a different penetration, but takes about 30 seconds instead of 12 seconds.

The running time is also sensitive to the order in which constants are declared, especially keys, since there are many situations where it tries the possible keys in the given order, and it finds the solution more quickly if the one it needs occurs early in the list.

2.10. *Protocol Specification Variations*

There are, in general, many ways to specify the same protocol, which are "correct" in some sense. Yet they lead to different running times, and some may exclude possible penetrations.

The use of symbolic constants is an area where choices are made. For example, the TMN protocol specification uses constants *a* and *b* to represent the initiating and listening parties, yet any two parties (out of a population of perhaps hundreds) could take on either role. By using constants *a* and *b*, rather than variables, to define the protocol, we are essentially using the mathematician's proof trick of saying "with no loss of generality, we may assume that ...." Since some party acts as initiator, and another as listener, we may as well call them "*a*" and "*b*." It is less clear, however, that some penetrations are not missed by failing to mention other parties "*c*," "*d*," etc., who might conceivably play a part in some subtle vulnerability. Dolev and Yao addressed this question in their paper, and proved that it was sufficient to consider only the "active" parties in the protocol, but their result was in the context of a particular model that is not general enough to cover the TMN protocol or others we must analyze.

Another simplification is that the state graph for each party is loop-free. In actual protocols there are various ways for a party to return to its initial state and begin a new conversation. While it is possible to specify loops in the state graph, they are avoided because they can cause the program to get into a loop.

Many protocol vulnerabilities are, in fact, made possible only by exploiting two or more separate conversations involving the same party. Simmons's penetration of the TMN protocol is an example of this. In this case, and in others, it is still possible to find the vulnerability with a nonlooping version of the protocol, because all sessions but one are passively observed; the penetrator eavesdrops on them but does not modify any message. In such cases it is sufficient to include the record of eavesdropped sessions as initial knowledge. However, it is not clear for what general class of protocols this tactic is adequate.

Until rigorous results are available for such questions, our simplifying tactics are justified in part by arguments applicable to particular protocols, or by the philosophy that we are accomplishing something useful by a systematic search for some vulnerabilities, even in the absence of a proof that no vulnerabilities are neglected.

3. The NRL Protocol Analyzer

In this section we describe the NRL Protocol Analyzer and the way in which it was used to analyze the TMN protocol [28]. The NRL Protocol Analyzer is an

interactive program written in Prolog [6] that can be used to assist either in the verification of security properties of cryptographic protocols or in the detection of security flaws. It is currently implemented in both Quintus Prolog and SWIProlog. Although it is still under development, it has already been used successfully to detect previously unknown flaws in the Simmons Selective Broadcast Protocol [26] and the Burns–Mitchell Resource Sharing Protocol [4]. The results of these analyses, using earlier versions of the Protocol Analyzer, are contained in [18 and 17].

In the case of the TMN protocol we were already aware of the security flaw by the time we used the tool to analyze it. Since the Protocol Analyzer is interactive and depends at least partly upon the intuition of the user for its guidance, the fact that we already knew how to break the protocol meant that it was not a complete test of the Analyzer's powers. Thus, the analysis presented in this paper does not present as strong evidence for the usefulness of the tool as those of the Simmons and Burns–Mitchell protocols. However, it can still be used as a means of illustrating how the Analyzer works. The simplicity of the TMN protocol means that we are able to give a more complete description of our analysis than we were in our descriptions of the analyses of the Simmons and Burns–Mitchell protocols, and thus we can give the reader a better idea of how the Analyzer operates.

3.1. *Description of the Model and Specification Language*

The model that we use takes the same approach as the term-rewriting model of Dolev and Yao [8]. In their model the intruder or intruders are assumed to have complete control of the communications network in that they are able to read all traffic, identify the source of each message, alter any message, destroy any message, and construct and initiate messages themselves. Thus, the honest participants in the protocol are communicating directly, not with each other, but with the intruder. We may thus think of the protocol as a machine that the intruder manipulates to produce messages. In the Dolev–Yao model a protocol is insecure if an intruder is able to manipulate it to produce one or more of a designated set of messages that are supposed to be secret.

The main difference between our model and the Dolev–Yao model is that the Dolev–Yao model treats a protocol as a machine for producing words, while our model treats a protocol as a machine for producing three things: words, beliefs, and events. In our model each participant in the protocol possesses a set of beliefs. Beliefs are modified or created as the result of receiving messages made up of words, while messages are sent depending upon both beliefs and messages received. Events represent the state transitions in which new words are generated and beliefs are modified. Thus an intruder who controls the dissemination of messages can use the protocol to produce words, beliefs, and events.

Since our analysis program is written in Prolog, we use several Prolog conventions in our specification language. Thus variables (for which it is possible to substitute other variables and expressions) are represented by capital letters or strings beginning with capital letters, while constants are represented by lowercase letters or strings beginning with lowercase letters. All variables appearing in a statement are universally quantified. Thus, a statement "if foo(X), then fie(X, Y)" should be interpreted to mean "for all X and Y, if foo(X), then fie(X, Y)."

A specification in the NRL Protocol Analyzer consists of four sections. The first section consists of transition rules governing the actions of honest principals. It may also contain rules describing possible system failures that are not necessarily the result of actions of the intruder, for example, the compromise of a session key. The second section describes the operations that are available to the honest principals and the users. The third section describes the atoms that are used as the basic building blocks of the words in the protocol. The fourth section describes the rewrite rules obeyed by the operations.

3.1.1. *Transition Rules.* A transition rule has three parts.

Pre-Conditions. The first part of a transition rule gives the conditions that must hold before the rule can fire. These conditions describe the words the intruder must know (that is, the message that must be received by the principal), the values of the state variables available to the principal (referred to as "lfacts" in the transition rule), and any conditions on the state variables and words.

Post-Conditions. The second part describes the conditions that hold after the rule fires in terms of words learned by the intruder (that is, the message sent by the principal) and any new values taken on by state variables. Each time a rule fires, a counter local to the principal is incremented; this is also recorded in the preconditions and postconditions of the rule.

Event Statements. The third part of the rule consists of an event statement. It is used to record the firing of a rule and is useful for indicating what the rule does. It is determined by the rest of the rule. The event statement describes a function with four arguments. The first gives the name of the relevant principal. The second gives the number of the protocol round. The third identifies the event. The fourth gives the value of the principal's counter after the rule fires. The value of the event is a list of words relevant to the event.

Here is a typical specification of a transition rule, describing a key server's receiving a request for a key, where R is supposed to be a nonce generated by user(A, honest). Note that R is simply a variable, since the intruder could have substituted any word for R. In this protocol the key server sends out a key and stores the user's request, together with the key.

```
rule(3)
If:
count(server) = N,
intruderknows([user(A,Y),user(B,X),R]),
then:
count(server) = s(N),
lfact(server,N,key,s(N)) = [user(A,Y),user(B,X),key(server,N)],
intruderlearns([e(key(user(A,Y),(user(B,X),key(server,N),R))]),
EVENT:
```

```
event(server,N,sentkey,s(N)) =
  [user(A,Y),user(B,X),R,key(server,N)].
```

Note that the principal involved in this transition, the server, is identified only as arguments of state variable assignments. This is in contrast to, for example, the Interrogator, which uses a similar model but which uses the more conventional notation of representing each principal's local state space as a list of words. The main reason for our choice of this nonstandard notation is to make it easier to allow queries of partial descriptions of states. If state variables are represented as separate function assignments, it is easy to ask questions about the reachability of partial descriptions by asking questions about the reachability of subsets of sets of assignments.

3.1.2. *Operations on Words.* The second section of the specification defines the operations that can be made by honest principals and by the intruder. If an operation can be made by the intruder, the Analyzer translates it into a transition rule similar to the above, except that the relevant principal is the intruder instead of an honest principal, and no lfacts are involved. An example of a specification of an operation is the following, describing public-key encryption:

```
fsd1:pke(X,Y):length(X)=1:length(pke(X,Y))=length(Y):pen.
```

The term "fsd" stands for "function symbol description." The next term gives the operation and the arguments. The third gives conditions on the arguments. In this case we make the condition that the key be a certain length, which in this case we make a default unit length one. The next term gives the length of the resulting word, which in this case is the length of Y. The last field is set to "pen" if the penetrator can perform the operation, and "nopen" if he cannot.

Some operations are built into the system. These are: concatenation of X and Y, denoted by (X, Y), taking the head (first element) of a list L, denoted by head(L), taking the tail (all but the first element) of a list L, denoted by tail(L), and id_check, which is used by an honest principal to determine whether or not two words are equal. The function id_check(X, Y) evaluates to "ok" when X and Y are equal.

3.1.3. *Specifying Atoms.* The third section describes the words that make up the basic building blocks. Examples would be user names, keys, and random numbers. Again, we indicate whether or not the word is known to the intruder in the last field of an atom specification. It is "known" if the intruder knows it, and "notknown" if the intruder does not know it.

3.1.4. *Specifying Rewrite Rules.* The last section describes the rewrite rules by which words reduce to simpler words. An example of a rewrite rule would be one which describes the fact encryption with corresponding public and private keys cancel each other out:

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

The Analyzer is queried by asking it to find a state that consists of a set of words known by the intruder, a set of lfacts, and a sequence of events that must have occurred. Conditions can be put on the words, lfacts, and events by putting conditions on the words that appear in them. Conditions can also be put on the results by specifying that certain sequences of events must *not* have occurred.

3.1.5. *Assumptions Underlying the Analyzer.* There are several assumptions underlying the Analyzer. We list them below:

1. A principal is either honest, that is, he or she follows all rules of the protocol, or he or she is dishonest. All dishonest principals are assumed to be in cooperation with the intruder; thus, any words known to a dishonest principal (such as master keys) should also be assumed to be known by the intruder. It is up to the protocol specifier to make the assumption whether or not dishonest principals exist.
2. The intruder can only learn a word or cause a state variable to change as a result of an application of a combination of the transition rules and the rewrite rules. Thus, for example, if the assumption that keys may be compromised under certain circumstances is desired, a rule explicitly describing key compromise will have to be written. This was done, for example, in our analysis of the Burns–Mitchell protocol.
3. When the intruder learns a word, he or she learns the entire significance of that word. Thus, when the intruder learns e(key(X), rand(X, N)) he or she knows that is the encryption of a random number generated by X encrypted with a key belonging to X, although he or she may not know key(X) or rand(X, N).
4. Two words cannot be identical unless they are identical syntactically. Thus key(X) can never be equal to rand(X, N), and rand(X, N) is not equal to rand(Y, M) unless $X = Y$ and $M = N$. If a specification designer wishes to allow for the possibility, say, that a randomly generated number will be equal to a key, this must be allowed for by the appropriate use of variables so that a substitution making the two words identical exists.

3.2. *Analyzing a Protocol*

The NRL Protocol Analyzer is similar to many other tools for analyzing communication protocols in that it makes use of specifications of protocols as communicating state machines, and that analysis is performed by proving that undesirable states are unreachable. However, there are a couple of important differences which mean that we could not apply conventional protocol analysis techniques.

The first important difference is that the search space is infinite. We assume, for example, that if a principal applies the same encryption function over and over, this will result in an infinite sequence of words. Although, in fact, repetition will eventually occur, in a well-designed cryptosystem this will not happen for a very long time. Since exhaustive search of so large a space is infeasible, we may assume, for all practical purposes, that it is infinite. (This is the same assumption made by

Dolev and Yao.) Thus techniques developed for exhaustive search do not give us much benefit.

Another important difference is that, in analyzing a protocol, it is not enough to ask what transition gives us output that is equal to the input we desire. Instead, for each output we need to ask what substitutions, if any, give us output that is *reducible* to the input we desire. Thus we have to include algorithms for determining such substitutions.

These differences from the assumptions underlying conventional communication protocol analysis mean that we needed to develop new techniques for analyzing protocols. In this section we outline what these techniques are and how they are used.

3.2.1. *Querying the Protocol Analyzer.* The Protocol Analyzer is used by specifying a set of states using terms built up of variables, constants, and function symbols. A state is specified by listing a set of words learned by the intruder and a set of lfacts belonging to the honest participants, along with a set of conditions on the words and lfacts. A set of sequences of events that must not have occurred or a sequence of events that must have occurred can also be listed.

The protocol analysis program takes each subset D of the desired words, beliefs, and events and compares it with each subset O of the words, beliefs, and events output by each rule. It then uses a variant of the narrowing algorithm of Rety *et al.* [24] to find a complete set of substitutions Σ to the variables in D and O, if any such exist, that leave D irreducible but make O reducible to D (where by "complete" we mean that if there is any other such substitution, it can be obtained by applying a further substitution to a substitution from Σ). Thus, if we are attempting to find all cases in which an intruder can learn a word $e(X, Y)$, and a rule with output $e(A, B)$ exists, the program will find the two substitutions:

1. $A = X, B = Y$, and
2. $B = d(A, e(X, Y))$.

The program then applies these substitutions to the input and output of the rule. For each such set of substitutions, it generates a result consisting of the words matched, the internal state values matched, the events matched, the events not matched; the event that occurred, the events that will occur in the future, the events that must be avoided, and the words and internal state values input into the rule. All words in the result are assumed to be in their reduced form.

Once we have this result, we can use the Analyzer to determine from what states the input state (i.e., the state defined by the state values and words that are given as input to the rule, and by the state values, words, and event statements not yet matched) is reachable, and via what events. This will allow us to keep a consistent picture of the history of events that leads to the state we are looking for.

If we apply this approach mindlessly, we will of course wind up with an explosion of states, and never reach our goal. Thus we must develop ways of limiting the search space. The Analyzer supports several ways of doing this.

3.2.2. *Partial Queries.* The first tool available to the user is the use of selection in state querying. When the user is presented with a state, he does not have to ask how the whole state can be reached; instead, he can ask how some piece of it is reachable. For example, in the encryption example given above, once the user is presented with the fact that the intruder can produce W if he knows X and $d(X, W)$, he does not have to ask the system how the intruder can find X and $d(X, W)$; instead he can ask how the intruder can find $d(X, W)$. If the intruder cannot find $d(X, W)$, then he certainly cannot find X and $d(X, W)$.

3.2.3. *The Language Checker.* The user is also given several tools to assist him in proving lemmas about the unreachability of states. The first of these tools we call the Language Checker. The Language Checker makes use of the fact that many of the infinite paths followed by the Analyzer define formal languages. For example, suppose that the user wants to find out how the intruder can find a word k, and that the Analyzer tells him that this can be done only if he can find $e(W, k)$, where W is any word. Suppose that the user presents $e(W, k)$ to the Analyzer, and it tells him that this can only be found if the intruder knows $e(W1, e(W, k))$, and so on. This pattern of results suggests the language **E** with the following productions:

1. $\mathbf{E} \rightarrow k$,
2. $\mathbf{E} \rightarrow e(\mathbf{W}, \mathbf{E})$,

where **W** is the language consisting of all computable words. If we can show that intruder knowledge of any irreducible word of **E** requires previous knowledge of an irreducible word of **E**, then we will have shown that the entire language **E** is unreachable. Thus, any state in which the intruder knows a word of **E** is unreachable.

The Language Checker can be used to show that a language is unreachable by showing that intruder knowledge of any word of it requires intruder knowledge of some other word of the language. After the Language Checker has been used to show that the language is unreachable, the language can be loaded into the Analyzer, which will then reject as unreachable any state that requires intruder knowledge of a word of that language.

3.2.4. *The State Unifier.* Another tool used by the Analyzer is the State Unifier. The State Unifier makes use of the fact that it is often possible to prove that a state is reachable only if it is in a certain form. For example, an honest protocol participant user(A, honest) may possess a local state variable that contains a random number generated by that user. We can easily prove that variable is set to W only if $W = \text{rand}(\text{user}(A, \text{honest}), N)$. The State Unifier can be used to find the appropriate conditions that make a state reachable and output them in a form readable by the Analyzer. The Analyzer can then use the results so that, whenever the specified state is returned, the appropriate substitutions can be made. Thus, for example, if the Analyzer produces a state in which the state variable described above is set to W, it will make the substitution $W = \text{rand}(\text{user}(A, \text{honest}), N)$ if possible. If the substitution is not possible, the state is rejected as unreachable. This prevents having to prove the same result over and over again every time that state is produced.

When the Analyzer is used to analyze a protocol, one generally begins by using the Language Checker and the State Unifier to reduce the search space. Once this

is done, the analysis of the protocol generally becomes rather straightforward. However, this means that there is a lot of overhead associated with the analysis. It is our intention to automate as much of this analysis as possible. We have already achieved part of our goal with the development of the Language Checker and the State Unifer. However, we intend to do more along those lines.

3.2.5. *Modes of Use.* Finally, we note that there are two modes in which the user can query the Analyzer. In one, the manual mode, the list of states that can immediately precede the queried state are presented to the user, the user queries each of these states to get the output that immediately precedes them, and so forth. Since the user can specify that only substates of each state be queried at each step, this mode makes it easier to keep the size of the search space under control. In the other, the automatic mode, the user only specifies the initial state, and the Analyzer itself queries all subsequently produced states, using breadth-first search. The automatic mode uses some simple tree-pruning heuristics (for example, empty lfacts and words known initially by the intruder are not queried) but in general it will query a larger portion of the state than a user might; thus greater ease of use is traded off against a larger search space.

Unreachable states and states that may be initial are identified as such when automatic mode is used. Note that a state can only be identified as potentially initial since the Analyzer does not always query the entire state. Thus, although a path may exist leading to some substate, the entire state may not be reachable, and thus the initial state in that path may not be truly initial. In this case the path must be re-examined manually.

It is possible to switch back and forth between automatic and manual modes. Thus, the user can start in automatic mode, and if the tree is becoming too bushy, then switch back into manual mode, erase part of the tree, and redo that part of the search in manual mode until the tree is of manageable size. The automatic mode can also be used to search only a part of the tree. The example that we present in this paper is so small that this was not necessary, but these techniques have proved helpful on larger problems.

3.3. *Description and Specification of the TMN Protocol*

We now describe the specification of the TMN protocol. As in our other specifications [18], [17], we divide the system into honest users who follow the rules of the protocol (identified by names of the form user(A, honest)) and dishonest users who are under the control of the intruder (identified by names of the form user(A, dishonest)). Each rule describes either the intruder communicating with an honest user of the system or the intruder generating words on his own.

Two kinds of encryption are used in the protocol: single-key encryption and public-key encryption. We assume that the same single-key and the same public-key algorithms are used throughout the protocol. Single-key encryption of word Y with key X is denoted by e(X, Y). Single-key decryption of word Y with key X is denoted by d(X, Y). Public keys are denoted by pk(S), where S is some identifier, and the corresponding private key is denoted by sk(S). Encryption or decryption in a

public-key system is denoted by pke(K, X), where K is the public or private key, respectively.

The reduction rules that we assume hold are as follows:

$$\mathrm{e}(X, \mathrm{d}(X, Y)) \rightarrow Y,$$

$$\mathrm{d}(X, \mathrm{e}(X, Y)) \rightarrow Y,$$

$$\mathrm{pke}(\mathrm{sk}(\mathrm{server}), \mathrm{pke}(\mathrm{pk}(\mathrm{server}), X)) \rightarrow X,$$

$$\mathrm{pke}(\mathrm{pk}(\mathrm{server}), \mathrm{pke}(\mathrm{sk}(\mathrm{server}), X)) \rightarrow X.$$

The first five rules of the specification correspond directly to the rules of the protocol.

We begin by describing an honest user (identified as user(A, honest)) attempting to establish communication with another user, who may be either honest or dishonest (identified as user(B, X)). User(A, honest) sends the message containing the encrypted random number and remembers both the number and the name of the individual it is attempting to establish communication with:

```
rule(1)
If:
count(user(A,honest)) = [N],
then:
count(user(A,honest)) = [s(N)],
intruderlearns([pke(pubkey(server),rand(user(A,honest),N)),
 user(A,honest),user(B,W)]),
lfact(user(A,honest),N,step1,s(N)) =
 [user(B,W),rand(user(A,honest),N)],
EVENT:
event(user(A,honest),N,initrequest,s(N)) =
 [user(B,W),rand(user(A,honest),N)].
```

Next, the server receives a request, which is identified as coming from a user, user(A, Y), who may be honest or dishonest, requesting to communicate with a user, user(B, X), who may be honest or dishonest. The server decrypts the third message field and stores it as the key-encryption key. It also sends a message to user(B, X) telling it that user(A, Y) wishes to communicate with it.

```
rule(2)
If:
count(server) = [N],
intruderknows([X,user(A,Y),user(B,W)]),
length(X) = 1,
then:
count(server) = [s(N)],
lfact(server,N,keyencryptkey,s(N)) =
 [pke(privkey(server),X),user(A,Y),user(B,W)],
```

```
intruderlearns([user(A,Y)])
EVENT:
event(server,N,storedkey,s(N)) = [user(A,Y),user(B,W),X]
```

When user(B, honest) receives the call from the server, it generates a random number as a session key and sends the result to the server encrypted with the server's public key. It also remembers the random number and the name of the individual (user(A, Y)), that it is attempting to communicate with.

```
rule(3)
If:
count(user(B,honest)) = [M],
intruderknows([user(A,Y)]),
then:
count(user(B,honest)) = [s(M)],
lfact(user(B,honest),M,step2,s(M)) =
  [user(A,Y),rand(user(B,honest),M)],
intruderlearns([pke(pubkey(server),rand(user(B,honest),M))]),
EVENT:
event(user(B,honest),M,genkey,s(M)) =
  [user(A,Y),rand(user(B,honest),M)].
```

When the server gets a response from user(B, X), it decrypts it using its private key, encrypts the result using the random number generated by user(A, Y), and sends the result to user(A, Y). It now no longer needs the information from user(B, X) and user(A, Y), and so it deletes it.

```
rule(4)
If:
count(server) = [M],
intruderknows([X]),
lfact(server,N,keyencryptkey,M) = [Z,user(A,Y),user(B,W)],
length(X) = 1,
then:
count(server) = [s(N)],
lfact(server,N,keyencryptkey,s(M)) = [],
intruderlearns([e(Z,pke(privkey(server),X))]),
EVENT:
event(server,N,sentkey,s(M)) = [user(A,Y),user(B,W),Z,X].
```

The word "key" in the message identifies it as containing an encryption key.

When user(A, honest) receives what it believes to be an encrypted key from the server, it decrypts it and assumes that the result is a session key good for communicating with user(B, X). Note that the protocol rules do not give user(A, honest) any grounds for believing that a message is an encrypted key.

```
rule(5)
If:
count(user(A,honest)) = [M],
intruderknows([R]),
lfact(user(A,honest),N,step1,M) = [user(B,W),X],
length(R) = 1,
then:
count(user(A,honest)) = [s(M)],
lfact(user(A,honest),N,seskey,s(M)) = [d(X,R)],
EVENT:
event(user(A,honest),N,getkey,s(M)) = [R,user(B,W),X].


rule(6)
If:
count(user(A,honest)) = [M],
lfact(user(A,honest),N,step1,M) = [user(B,W),X],
lfact(user(A,honest),N,seskey,M) = [K],
then:
count(user(A,honest)) = [s(M)],
EVENT:
event(user(A,honest),N,acceptkey,s(M)) = [user(B,W),K].
```

Rule 6 is somewhat artificial; it is put in because we need a transition in which user(A, honest) accepts the key. Usually, user(A, honest) would accept the key as the result of doing some sort of checking. However, in this protocol user(A, honest) accepts the key as genuine without doing any checking. Thus this transition is empty.

Finally, we include the various rules that are used to define atoms, function symbols, and rewrite rules:

```
fsd1:e(X,Y):length(X)=1:length(e(X,Y))=length(Y):pen.
fsd2:d(X,Y):length(X)=1:length(d(X,Y))=length(Y):pen.
fsd3:pke(X,Y):length(X)=1:length(pke(X,Y))=length(Y):pen.


        atom1:rand(user(A,dishonest),N):1:known.
        atom2:rand(user(A,honest),N):1:notknown.
        atom3:privkey(server):1:notknown.
        atom8:pubkey(server):1:known.


rr1: e(X,d(X,Y)) => Y.
rr2: d(X,e(X,Y)) => Y.
ee3: pke(privkey(server),pke(pubkey(server),Y)) => Y.
rr4: pke(pubkey(server),pke(privkey(server),Y)) => Y.
```

3.4. *Analysis of the TMN Protocol*

In analyzing the protocol we attempted to determine how it was possible for the system to reach a state in which user(A, honest) believed that some word K was a key, and K was known to the intruder.

Before we began our analysis, we used the Language Checker and the State Unifier to limit the search space. We defined six languages and used the Language Checker to show that they were unreachable. Lack of space prevents us from going into detail about how these languages were specified, but, to give an idea of the flavor of the kinds of things we proved, one of the languages was used to prove that the intruder could never learn the server's private key; another was used to prove that the intruder could not learn any word of the form d(X, Y) unless he already knew Y.

We also defined three state condition specifications for the State Unifier. These give the conditions under which a state is reachable. The states defined were: the state in which the intruder knows a word of the form e(X, Y) without knowing Y (Y must be of the form pke(privkey(server), W)), the state in which the intruder knows a word of the form pke(X, Y) without knowing Y (Y must be of the form pke(pubkey(server), rand(user(B, honest), N))), and the state in which an honest user's lfact step1 is nonempty (it must be set to [user(B, H), rand(user(A, honest), N)]). The first two definitions were constructed using the Language Checker; the third was constructed by querying the state directly and observing under what conditions the state was reachable.

Once the language and state files were generated, they were loaded into the Analyzer. Now, when a user queried the Analyzer on how to find a state, it would find the set of states that could immediately precede it and, for each such state, determine whether or not it contained a word of an unreachable language. If it did, that state was discarded. If the state contained a set of words and lfacts described in the state file, the Analyzer attempted to modify them so that they satisfied the conditions specified in the state file. If they could be so modified, that state was discarded. In this way we could keep the size of the search space under control.

Once we had done this initial syntactic analysis, we were ready to use the Analyzer to look for attacks that allowed an intruder to find out a session key. The most obvious way was to begin by asking it if there were any states in which the intruder knew a word and some honest user user(A, honest) had accepted that word as a key for communication with another honest user. (This is trivially the case for communication with a dishonest user, since all dishonest users are assumed to share all information with the intruder.) If we did this, however, we would have generated an unmanageably large search space. This is because the Analyzer will attempt to satisfy the two subgoals separately as well as together, and there are a very large number of ways an intruder can find out a single word with no conditions put on it. What we decided to do instead was first to find out the conditions under which user(A, honest) would accept a word as a key. Knowing those conditions would restrict the set of words that the Analyzer would have to look for when we gave it the full query.

As it turned out, in our attempt to determine under what conditions a word may

be accepted as a key, we found an attack in which the intruder passes off a word generated by himself as a session key. We used the automatic query option, so that each state produced by the Protocol Analyzer was queried by the Analyzer until only initial or unreachable states were left. The output ran to about two pages; we give the beginning and the end here:

```
5 ?- autoquery.
What state do you wish to query?
|:

What words is the intruder looking for?
|:
What state variable values is the intruder looking for?
|:

List the sequence of events that you want to have occurred.
|: event(user(A,honest),N,acceptkey,M) = [user(B,honest),K]
|:
What conditions to you want to put on all of these?
|:

List the sequences of events that you dont want to have occurred.

Enter a list
|:

Specify a range of rule numbers that you want
|:


Solution number 1
The events that occurred are
    R1 = event(user(G4465, honest), [G4468], acceptkey,
 s(G4471)) =
   [user(G4476, honest), G4479].
Input state variables are:
    S1 = count(user(G4465, honest)) = G4471.
    S2 = lfact(user(G4465, honest), [G4468], step1, G4471) =
 [user(G4476, honest), rand(user(G4465, honest), G4468)].
    S3 = lfact(user(G4465, honest), [G4468], seskey, G4471) =
 [G4479].
Rule number 6 was used.
querying state 1
Found state 1.1
Found state 1.2
querying state 1.1
Found state 1.1.1
querying state 1.2
Found state 1.2.1
```

```
Found state 1.2.2
Found state 1.2.3
querying state 1.1.1
The state 1.1.1 may be an initial one.
querying state 1.2.1
.
.
.
querying state 1.2.2.2.7.1
The state 1.2.2.2.7.1 may be an initial one.
querying state 1.2.2.2.7.2
The state 1.2.2.2.7.2 is unreachable.
querying state 1.2.2.3.1.1
The state 1.2.2.3.1.1 may be an initial one.
querying state 1.2.2.3.1.2
The state 1.2.2.3.1.2 is unreachable.
querying state 1.2.2.4.1.1
The state 1.2.2.4.1.1 may be an initial one.
querying state 1.2.2.4.1.2
The state 1.2.2.4.1.2 is unreachable.
```

We then looked at each path beginning in a possible initial state. Several of these described attacks. A typical one was the path beginning in state 1.2.2.4.1.1. This generated the following path:

```
Solution number 1.2.2.4.1.1
The events that occurred are
    R1 = event(user(G264, honest), [G267], initrequest,
 s(G267)) =
   [user(G279, honest), rand(user(G264, honest), G267)].
Input state variables are:
    S1 = count(user(G264, honest)) = G267.
States found are:
    D1 = lfact(user(G264, honest), [G267], step1, s(G267)) =
 [user(G279, honest), rand(user(G264, honest), G267)].
Words found are:
    E1 = pke(pubkey(server), rand(user(G264, honest), G267))
Rule number 1 was used.

Solution number 1.2.2.4.1
The events that occurred are
    R1 = event(server, [G340], storedkey, s(G340)) =
 [user(G350, G351), user(G356, G357),
  pke(pubkey(server), rand(user(G264, honest), G267))].
Input words are:
    W1 = pke(pubkey(server), rand(user(G264, honest), G267))
```

```
    W2 = user(G350, G351)
    W3 = user(G356, G357)
Input state variables are:
    S1 = count(server) = G340.
    S2 = lfact(user(G264, honest), [G267], step1, s(G267)) =
 [user(G279, honest), rand(user(G264, honest), G267)].
States found are:
    D1 = lfact(server, [G340], keyencryptkey, s(G340)) =
 [rand(user(G264, honest), G267),
  user(G350, G351), user(G356, G357)].
Rule number 2 was used.

Solution number 1.2.2.4
The events that occurred are
    R1 = event(pen, [G385], pen_pke, s(G385)) =
 [pubkey(server), G282].
Input words are:
    W1 = pubkey(server)
    W2 = G282
Input state variables are:
    S1 = count(pen) = G385.
    S2 = lfact(server, [G340], keyencryptkey, s(G340)) =
 [rand(user(G264, honest), G267),
  user(G350, G351), user(G356, G357)].
    S3 = lfact(user(G264, honest), [G267], step1, s(G267)) =
 [user(G279, honest), rand(user(G264, honest), G267)].
Words found are:
    E1 = pke(pubkey(server), G282)
Rule number 303 was used.

Solution number 1.2.2
The events that occurred are
    R1 = event(server, [G340], sentkey, s(s(G340))) =
 [user(G350, G351), user(G356, G357),
  rand(user(G264, honest), G267), pke(pubkey(server), G282)].
Input words are:
    W1 = pke(pubkey(server), G282)
Input state variables are:
    S1 = count(server) = s(G340).
    S2 = lfact(server, [G340], keyencryptkey, s(G340)) =
 [rand(user(G264, honest), G267),
  user(G350, G351), user(G356, G357)].
    S3 = lfact(user(G264, honest), [G267], step1, s(G267)) =
 [user(G279, honest), rand(user(G264, honest), G267)].
Words found are:
    E1 = e(rand(user(G264, honest), G267), G282)
Rule number 4 was used.
```

```
Solution number 1.2
The events that occurred are
    R1 = event(user(G264, honest), [G267], getkey,
 s(s(G267))) =
   [e(rand(user(G264, honest), G267), G282),
   user(G279, honest), rand(user(G264, honest), G267)].
Input words are:
    W1 = e(rand(user(G264, honest), G267), G282)
Input state variables are:
    S1 = count(user(G264, honest)) = s(G267).
    S2 = lfact(user(G264, honest), [G267], step1, s(G267)) =
 [user(G279, honest), rand(user(G264, honest), G267)].
States found are:
    D1 = lfact(user(G264, honest), [G267], seskey,
 s(s(G267))) =
   [G282].
Rule number 5 was used.

Solution number 1
The events that occurred are
    R1 = event(user(G264, honest), [G267], acceptkey,
 s(s(s(G267)))) =
   [user(G279, honest), G282].
Input state variables are:
    S1 = count(user(G264, honest)) = s(s(G267)).
    S2 = lfact(user(G264, honest), [G267], step1,
 s(s(G267))) =
   [user(G279, honest), rand(user(G264, honest), G267)].
    S3 = lfact(user(G264, honest), [G267], seskey,
 s(s(G267))) =
   [G282].
Rule number 6 was used.
```

The attack is as follows. In the first transition user(G264, honest), intending to communicate with user(G279, honest), encrypts his random number with the server's public key, following the first step of the protocol. In the second the server receives the message, following the second step of the protocol. (Note however, that the user names received are two entirely new ones. This is because the Analyzer is giving us the most general conditions under which the transition may take place; any two user names will suffice.) In the third, however, the intruder encrypts some word that he already knows with the server's public key. In the fourth step the intruder, impersonating user(G279, honest), sends the encrypted word to the server. The server decrypts the word, encrypts it with the random number from user(G264, honest), and sends it to user(G264, honest). In the fifth and sixth steps user(G264, honest) decrypts the message with his random number and accepts the word generated by the intruder as a key.

3.5. *Discussion*

By using the Protocol Analyzer, we were able to "discover" one of the flaws in the TMN protocol. However, we did not find the flaw discovered by Simmons. This was because we did not model the algebraic properties of RSA and modular addition that were necessary in order to uncover that flaw.

Our choice not to model these properties was a result of the fact that the narrowing algorithm used by the Protocol Analyzer only works for rewrite rules. Both RSA and modular addition involve commutative operations which cannot be expressed as rewrite rules. Although the particular properties of RSA and modular addition that led to the flaw *could* be written as rewrite rules, we could only guess which rules to choose after we knew the flaw; thus a demonstration of the use of the tool in finding these flaws would not have been very useful.

There are several ways in which the Protocol Analyzer could be improved that we are currently investigating. The first of these is the use of unification algorithms that work for more general equational theories than rewrite rules. In the area of unification algorithms, we are investigating the use of algorithms that combine unification algorithms for several theories [3]. Such algorithms would allow us to consider cases in which some operators obey rewrite rules and others the laws, say, of an Abelian group or a Boolean ring. Since much of the complexity of the algebraic identities used in cryptographic protocols arises from the use of combinations of different operators (such as single-key encryption and exclusive-or) that obey different sets of algebraic rules for which efficient unification algorithms exist, this seems to be the most promising avenue to pursue. We note, however, that equational theories for which unification is undecidable exist. Thus it is possible that we may encounter operators that obey a set of algebraic rules for which no unification algorithm can be found.

We are also investigating improvements to the user interface and the automated support offered by the tool. At this point, the main drawback to the user interface offered by the tool is that a large amount of data describing different paths through the protocol is output that is difficult to manage and interpret. We are currently investigating better ways of displaying and managing this data. In the area of automated support, we find that the preliminary syntactic analysis is still relatively difficult and time consuming, largely because it is still the user's responsibility to generate the lemmas that the Analyzer proves. One way to make the task easier would be to have the Analyzer generate the lemmas automatically, possibly with some assistance from the user. This seems to be feasible, since in most cases the generation of lemmas is a tedious repetitive process that could probably be automated. Another area in which the automated assistance could be improved is in the automated query mode. The introducton of more sophisticated tree-pruning algorithms would allow us to reduce the frequency with which the user has to switch back to manual mode, and thus make the search easier.

Finally, we are investigating improved ways of specifying the results the Analyzer proves for us. The Analyzer can be used to prove that an insecure state is unreachable, but it does not give us any assistance in deciding what states are the insecure ones. Thus, although using the Analyzer gives us added confidence that certain security flaws do not exist, more is needed before we can extend that

confidence to a confidence that a protocol is "secure" in general. What is needed is a uniform language for specifying security requirements so that they can be analyzed independently of the protocol. We have been developing such a language. It uses requirements on sequences of events to specify security properties of a protocol. These events can be mapped to the event statements in the protocol transitions. The language and how it is used is described in more detail in [27].

4. Inatest Experience

4.1. *Introduction*

The approach presented in this section analyzes encryption protocols using machine-aided formal verification techniques. The idea of the approach is to specify formally the components of the cryptographic network and the associated cryptographic protocol rules or actions. The formal method used is the Formal Development Methodology (FDM), which is an example of the state machine approach to formal specification. When using the state machine approach a system is viewed as being in various states. One state is differentiated from another by the values of state variables, and the values of these variables can be changed only via well-defined state transitions. Thus, the components are represented as state constants and variables, the protocol rules are represented as state transitions, and assumptions about the cryptographic algorithms are specified as axioms. The desirable properties that the protocol is to preserve are expressed as state invariants and the theorems that must be proved to guarantee that the system satisfies the invariants are automatically generated by the verification system. The formal specifications can also be tested by symbolically executing the formal specifications, which is what was done for the analysis of the TMN protocol presented in this section.

The formal specification language that is used is a variant of Ina Jo,[1] which is a nonprocedural assertion language that is an extension of first-order predicate calculus. The key elements of the Ina Jo language are types, constants, variables, axioms, definitions, initial conditions, criteria, and transforms. A criterion is a conjunction of assertions that specify the critical requirements for a good state (i.e., a secure state). A criterion is usually referred to as a state invariant since it must hold for all states including the initial state. An Ina Jo language transform is a state transition function; it specifies what the values of the state variables will be after the state transition relative to their values before the transition. The system being specified can change state only as described by one of the state transforms. A complete description of the Ina Jo language can be found in the Ina Jo Reference Manual [25].

4.2. *Formal Specification of the TMN Protocol*

In this section the important aspects of the Ina Jo specification for the TMN protocol are discussed. The complete specification is presented in the Appendix.

[1] Ina Jo is a trademark of Unisys.

Each user has a table of pending keys and a table of current keys.[2] These are represented by the Ina Jo *state variables*

Pending_Key(User, User): Key_Type,
Current_Key(User, User): Key_Type,

where Current_Key($U1$, $U2$) is user $U1$'s current session key for communicating with user $U2$, and similarly for Pending_Key.

The Server also has a table that is used for retaining keys during the key establishment protocol that are to be used to encrypt the new session key when returning it to the requester. This table is represented in the specification by the variable

In_Process(User, User): Key_Type.

There are three other state variables in the specification. They are

Net: Messages,
Keys_Used: Keys,
Intruder_Info: Information.

The first represents the network itself, the second all the keys that have ever been used in the network (this includes keys that are used for establishing a session key as well as the session keys themselves), and the information known by the intruder.

Before discussing the details of the Ina Jo transforms it is necessary to say a few words about some of the types and constants in the specification. Communication in the network is via messages, where a message is composed of a type, a source, a destination, and contents. The field Type is the only field of the message header that is explicitly represented in the specification; message address fields are not included since all users can retrieve all messages in a broadcast network. Source is used to indicate the user that requested a new session key, and Destination indicates the user with whom the requesting user wants to communicate. The Ina Jo specification *constants*

Type(Message): Message_Type,
Source(Message): User,
Destination(Message): User,
Contents(Message): Text

are used to retrieve each of the relevant parts of a message. They are constants because unlike state variables they do not change value from state to state. That is, for any given message they will always produce the same values.

The contents of messages sent from a user to the Server are encrypted using a public-key encryption algorithm and the Server's public key. These encryption

[2] It is sufficient to have a single table if it is assumed that a user will never attempt to establish a new session key with another user for which a session key currently exists.

algorithms and the Server's keys are specified using the following constants:

Server_Public: Key,
Server_Secret: Key,
PEncrypt(Key, Text): Text,
PDecrypt(Key, Text): Text

Finally, the Server uses a conventional encryption algorithm to encrypt the contents of messages for the users, and this algorithm is also used for user-to-user communication. These conventional algorithms are represented in the specification by the constants

Encrypt(Key, Text): Text,
Decrypt(Key, Text): Text.

As Massey pointed out in his presentation at the Oberwolfach Workshop [16], it is essential, when proving the security of an information system, to state any assumptions that are being made. In the case of an encryption protocol, these assumptions should include: what the intruder knows about the system; what messages the intruder observes; and what other actions the intruder is assumed to be able to perform. A "clear definition of security" must be presented as well.

All of the points raised by Massey are addressed in the TMN formal specification. What the intruder knows is modeled as Intruder_Info, what messages the intruder observes is explicitly expressed as part of the transforms, and the other actions that the intruder can perform are represented as additional transforms. Finally, the criterion part of the formal specification clearly presents the definition of security. In addition, assumptions about the encryption algorithms are presented as Ina Jo axioms.

For the TMN protocol specification there are five Ina Jo *axioms*. The first axiom

AXIOM

$$\forall t\text{: Text}(\text{PDecrypt}(\text{Server\_Secret}, \text{PEncrypt}(\text{Server\_Public}, t)) = t)$$

states that text encrypted using the public-key encryption algorithm and the Server's public key can be retrieved in the clear by using the public-key decryption algorithm and the Server's secret key. The second axiom

AXIOM

$$\forall t1, t2\text{: Text}, k\text{: Key}(\text{Times}(\text{PEncrypt}(k, t1), \text{PEncrypt}(k, t2)) = \text{PEncrypt}(k, \text{Times}(t1, t2)))$$

states that the public-key encryption algorithm distributes over the multiplication operator (Times in the specification).

The third axiom defines the division operator to be the inverse of the multiplication operator.

AXIOM

$$\forall t1, t2\text{: Text}(\text{Divide}(\text{Times}(t1, t2), t1) = t2).$$

The fourth axiom

AXIOM

$$\forall k, t\text{: Text}(\text{Decrypt}(k, \text{Encrypt}(k, t)) = t)$$

states that text encrypted using the conventional encryption algorithm can be recovered in the clear if it is decrypted using the conventional decryption algorithm and the same key.

The last axiom expresses the commutativity of the conventional encryption algorithm. That is,

AXIOM

$$\forall k1, k2\text{: Key}(\text{Encrypt}(k1, k2) = \text{Encrypt}(k2, k1)).$$

The second and fifth axioms are the critical axioms for demonstrating the flaw, although all five are used in the demonstration scenario.

The critical requirements that the system is to satisfy in all states are expressed in the *criterion* clause of the formal specification. For the TMN protocol the criterion states that the only key that is known to the intruder (i.e., a key contained in the set Intruder_Info) and that is also a key that was used by the system (i.e., a key in the set Keys_Used) is the Server's public key. Note that this includes keys used in the past as well as those presently being used. The criterion is expressed as follows:

$$\forall k\text{: Key}((k \in \text{Keys\_Used} \;\&\; k \in \text{Intruder\_Info}) \rightarrow k = \text{Server\_Public}).$$

The *initial* clause describes the requirements that must be satisfied when the system is initialized. For this system the initial state requires that there are no keys pending, there are no current keys for communicating with other users, and no key requests are in progress. All of these are indicated in the initial condition clause by using the special key value Null_Key. In addition, there are no messages in the network and the only keys that have been used are the Server's public key and secret key. Finally, the intruder starts out knowing the Server's public key (which is known to all users, since that is how they send messages to the Server).

The rules of the protocol being analyzed, which were presented in Section 1.3, are specified as Ina Jo *transforms*. There are five transforms that correspond to the normal state changes of the protocol. They are:

- Request_Key(A, B: User, K: Key), which corresponds to rule 1.
- Process_Request(A, B: User, C: Text), which corresponds to rule 2.
- Respond_To_Server(A, B: User, $R2$: Key), which corresponds to rule 3.
- Return_Key(A, B: User, C: Text), which corresponds to rule 4.
- Get_Key(A, B: User, C: Text), which corresponds to rule 5.

The specification does not contain transforms corresponding to the subsequent communication between users A and B. However, this would be done if a more complete analysis of the protocol were attempted.

The flaw that is demonstrated in this section is a generalization of the Simmons flaw, which relies on the fact that the RSA algorithm is homomorphic with respect to multiplication and that the conventional key algorithm is commutative. To model the intruders the specification also includes two user constants, Cheater and

Partner, which represent the cooperating intruders. The jointly agreed upon key that is used by the intruders is modeled by the constant Partner_Key. This key is returned whenever the server requests a key for communication with the Cheater from the Partner. There are also additional transforms that model the actions of the intruders. The three additional transforms are:

- Cheater_Request($R1$: Key), which is like the normal Request_Key transform except that the Cheater uses a key that is based on a previously encrypted key (PEncrypt(Server_Public, $R1$)) that the Cheater had intercepted.
- Partner_Response, which is like the Respond_to_Server transform except that the Cheater's partner always responds with the same key (Partner_Key), which the Cheater also knows.
- Compromise_Key(C: Text), which is analogous to the Get_Key transform except that the Cheater uses his pending key and the previously agreed upon key (Partner_Key) to compromise the encrypted key and the resulting information is used to update what the intruder knows (Intruder_Info).

Because the Partner_Key is a key known to the intruder that will also be used by the system, it is necessary to weaken the criterion to state that the only keys that can be known to the intruder and also used by the system are the Server's public key and the Partner_Key. Thus, the criterion is expressed as follows:

$$\forall k\text{: Key}((k \in \text{Keys\_Used} \;\&\; k \in \text{Intruder\_Info}) \rightarrow k \in \{\text{Server\_Public, Partner\_Key}\}).$$

Similarly, the initial clause must also specify that Partner_Key is initially known to the intruder.

Due to space limitation, only the Request_Key transform is discussed in detail in this paper. The Ina Jo transform for Request_Key is

```
Transform Request_Key(A, B: User, R1: Key)
Refcond
  R1 ∉ Keys_Used
Effect
  ∀U1, U2: User(
      N'' Pending_Key(U1, U2) =
            if U1 = A & U2 = B
                then R1
                else Pending_Key(U1, U2)
            fi)
& N'' Keys_Used = Keys_Used ∪ {R1}
& ∃m: Message(
      N'' Net = Net ∪ {m}
    & Type(m) = Request
    & Source(m) = A
    & Destination(m) = B
    & Contents(m) = PEncrypt(Server_Public, R1))
    & N'' Intruder_Info = Intruder_Info
        ∪ {PEncrypt(Server_Public, R1)}
```

The *Refcond* part of the transform expresses the conditions that must hold for the transition to take place. For the Request_Key transform this condition is that the key that A sends to the Server must not have been previously used. This is an example of a condition that is not likely to show up in the implementation, for it would be impractical for all users to know all of the keys that had been used. However, it is not unreasonable to assume this in the specification, for by using a pseudorandom number generator scheme to produce new keys the probability of coincidentally choosing an already used key is extremely low.

The resultant state after the state transition occurs (i.e., after the transform fires) is expressed in the *Effect* part of the transform. The $N''x$ notation is used in an Ina Jo specification to indicate the value that variable x has in the state that results from firing a transform. In the Effect section variables not preceded by N'' represent the value the variable had in the immediately preceding state. That is, in the state where the transform was fired. After the Request Key transform fires, User A's pending key for communicating with user B is $R1$, and $R1$ is added to the set of keys that have been used. A request message with source A and destination B is placed in the network and its contents is the key $R1$ encrypted using the public-key encryption algorithm with the Server's public key. Because the intruder is assumed to be able to listen passively to all network traffic the intruder information is enhanced with the encrypted form of $R1$.

Any state variables that do not appear in the effects section are assumed not to change. That is, the Ina Jo processor will automatically conjoin the expression

$$N''x = x$$

to the effect expression for any variable x that is not explicitly mentioned as changing in the effects section of the transform. This expression states that the new value of state variable x is equal to its old value. Therefore, for the Request Key transform the Ina Jo processor conjoins

$$\begin{aligned}&\forall U1, U2: \text{Text}(\\ &\quad N'' \text{ In\_Process}(U1, U2) = \text{In\_Process}(U1, U2)\\ &\quad \& N'' \text{ Current\_Key}(U1, U2) = \text{Current\_Key}(U1, U2))\end{aligned}$$

to the effect expression whenever it uses the effect section to generate a proof obligation.

The other transforms for this specification are interpreted in a similar manner.

4.3. *Formally Verifying the Specification*

After the formal specification is completed the theorems that are generated to check if the critical requirements (Ina Jo criteria) are satisfied can be verified. If the theorems are verified and the encryption algorithms satisfy the assumed axioms, then the system will satisfy its critical requirements.

The Formal Development Methodology employs an inductive approach to generate the necessary proof obligations to assure that the critical requirements are preserved. First, it must be shown that the criteria hold in the initial state. Next, for every transform it is necessary to show that if the transform fires in a state where

the criteria hold, then the resultant state also satisfies the criteria and the previous and new states satisfy the relationships expressed by the constraints. That is, the initial state is the basis case and the induction is on the transforms. Thus, the transforms can be fired in any order and by induction any reachable state will satisfy the criteria and any two consecutive states will satisfy the constraints.

The first proof obligation that is generated by the Ina Jo processor is the Initial Conditions Theorem:

$$\text{INIT} \rightarrow \text{CR},$$

where INIT is the INITIAL clause and CR is the criteria in the CRITERION clause of the specification.

In addition, for each transform in the specification the Ina Jo processor generates a Transform Theorem:

$$\text{CR} \ \& \ \text{R} \ \& \ \text{E} \rightarrow N'' \ \text{CR} \ \& \ \text{CO},$$

where R and E are the Refcond and Effect, respectively, for the transform, and CO is the CONSTRAINT clause.

Because the axioms represent the properties that the encryption algorithms are to satisfy, the system with a different encryption scheme can be verified by replacing the current axioms with axioms that express the properties of the new encryption scheme.

An advantage of expressing the system using formal notation and attempting to prove properties about the specification is that if the generated theorems cannot be proved the failed proofs often point to weaknesses in the system or to an incompleteness in the specification. That is, they often indicate the additional assumptions required about the encryption algorithm (i.e., missing axioms), weaknesses in the protocols, or missing constraints in the specification.

Because the analysis of this protocol was posed as a challenge problem and it was already known that the protocol was flawed, no attempts were made to prove the protocol. The interested reader can refer to [13] for more information on proving Ina Jo specifications and to [12] for more information on how failed proofs of protocol specifications can lead to discovering weaknesses in the protocol.

4.4. *Analyzing the TMN Protocol Specification*

The Simmons flaw in the TMN protocol was demonstrated using a specification execution tool for the Ina Jo language called Inatest [9]. The Inatest tool consists of a YACC program for translating the Ina Jo formal specification language into Lisp, and a set of utility routines written in Lisp that provide an environment suitable for symbolic execution. The Inatest system was originally implemented in Franz Lisp for the UNIX operating system running on a Digital Equipment Corporation VAX/750 or 780. The current version of the tool has been ported to Kyoto Common Lisp and runs on SUN workstations.

The Inatest symbolic execution tool provides a testing environment that allows the user to use various modes of operation to test formal specifications written in Ina Jo. The user submits a formal specification to the tool which then allows the user to interactively direct the tool to execute specified transforms symbolically.

For purposes of testing formal specifications a functional requirement may be thought of as a test case for the specification. Each test case consists of a start predicate, a sequence of operations (transforms) to be executed, and an optional desired resultant predicate to be compared with the actual result state. The start predicate specifies the assumptions about the state of the system before invoking any operation. The Inatest tool provides the user with several methods for defining the start predicate. It may be read in from a file, keyed in from the terminal, or the default start state may be used, which assumes the specification's initial predicate is the start predicate. In a similar manner the sequence of transforms to be executed and the resultant predicate may be read from a file or keyed in. The user also has the option to decide interactively what transform to execute next and with which actual parameters.

After executing a transform the user may display the current, start, or desired resultant predicate, list the available transforms, or list the specification being tested. The user may also change the predicate defining the current state by adding a predicate or by defining a new start predicate. The tool also provides a path command that allows the user to display the transforms that have been executed since the start predicate was defined as well as any assumptions that the user may have made along the way. The Inatest user may also save the current state of the execution for further execution at a later time. Saved states may be restored in any order without affecting the other saved states. A sample of the Inatest commands that are available to the user is given in the following table:

Add	—add a predicate
Exec [trans]	—execute transform "trans"
FIle [fileid]	—read commands from the named file
Help	—display available commands
LS	—list specification
LT	—list transforms
Path	—display current path
Quit	—return to unix
Restore [*N*]	—restore state number "*N*"
Save [comment]	—save a state
SEQ [fileid]	—execute a sequence of transforms
STATES	—display saved states
Vars [id]	—display [one] variable value(s)
Check [current *or* Result]	
Display [current *or* Start *or* Result]	
Init [start *or* Result]	

To test the Simmons flaw in the TMN protocol the default *start state*, which is the TMN specification's initial predicate, is used.

$$
\begin{aligned}
&\forall u1, u2\text{: User(} \\
&\quad \text{Current\_Key}(u1, u2) = \text{Null\_Key} \\
&\quad \& \ \text{Pending\_Key}(u1, u2) = \text{Null\_Key} \\
&\quad \& \ \text{In\_Process}(u1, u2) = \text{Null\_Key)} \\
&\& \ \text{Net} = \text{EMPTY} \\
&\& \ \text{Keys\_Used} = \{\text{Server\_Public, Server\_Secret}\} \\
&\& \ \text{Intruder\_Info} = \{\text{Server\_Public, Partner\_Key}\}.
\end{aligned}
$$

The sequence of transforms executed and the actual parameters used to demonstrate the flaw are

Request_Key(A, B, $R1$)
Cheater_Request($R1$)
Process_Request(Cheater, Partner, t)
  where t = Times(PEncrypt(Server_Public, Cheater_Key),
    PEncrypt(Server_Public, $R1$))
Partner_Response
Return_Key(Cheater, Partner, $t2$)
  where $t2$ = PEncrypt(Server_Public, Partner_Key)
Compromise_Key($t3$)
  where $t3$ = Encrypt(PDecrypt(Server_Secret, t),
    PDecrypt(Server_Secret, $t2$)).

The desired *resultant state* requires that key $R1$ that is sent by user A to the Server to start the protocol be part of the intruder's information. This requirement is expressed as

$$R1 \in \text{Intruder\_Info},$$

which is a clear violation of the security requirement since $R1$ is one of the keys used by the system.

A complete transcript of the Inatest session that demonstrated this flaw can be found in [14]. An overview of this session is presented in the following paragraphs.

The first transform invoked is Request_Key. Since the only keys used at the initialization time are the Server's public and secret keys,

$$R1 \notin \{\text{Server\_Secret}, \text{Server\_Public}\},$$

the Request_Key transform fires successfully and as a result

$$\text{PEncrypt}(\text{Server\_Public}, R1)$$

becomes part of the intruder's information. That is,

$$\text{Intruder\_Info} = \{\text{PEncrypt}(\text{Server\_Public}, R1), \text{Server\_Public}, \text{Partner\_Key}\}.$$

Next the Cheater_Request transform is invoked with parameter $R1$. Because the encrypted form of $R1$ is part of the intruder's information this transform fires successfully. The results of this transform firing that are of interest are that a request message is placed in the network requesting a session key for communication between the Cheater and the Partner, and the Contents field of this message contains the value

$$\text{Times}(\text{PEncrypt}(\text{Server\_Public}, \text{Cheater\_Key}), \text{PEncrypt}(\text{Server\_Public}, R1)),$$

which the intruder produces by multiplying the encrypted version of the Cheater_Key times the encrypted form of the key $R1$, which was retrieved by listening on the network.

In order to manage the expressions being generated more easily the Add

command is used to add the following information to Inatest's knowledge base:

$$t = \text{Times}(\text{PEncrypt}(\text{Server\_Public}, \text{Cheater\_Key}), \text{PEncrypt}(\text{Server\_Public}, R1)).$$

That is, this predicate is used to define an identifier for the expression.

Next the Process_Request transform is invoked with the parameters Cheater, Partner, and t. When executing this transform the cheater's request for the establishment of a session key is treated like any other user's request. That is, the In_Process table entry for the pair (Cheater, Partner) is set to the decryption of the contents part of the message

$$\text{PDecrypt}(\text{Server\_Secret}, t).$$

The Server also places a request message in the network for the cheater's partner:

$$\begin{aligned}&\exists m\text{: Message}(\\ &\quad N'' \text{ Net} = \text{Net} \cup \{m\}\\ &\&\ \text{Type}(m) = \text{Server\_Request}\\ &\&\ \text{Source}(m) = \text{Cheater}\\ &\&\ \text{Destination}(m) = \text{Partner}).\end{aligned}$$

The Partner_Response transform is used by the cheater's partner to respond to the Server's request. However, instead of sending a new key the partner sends the Server the key that the cooperating intruders have agreed upon (Partner_Key) as the new session key. This is done by placing a response message in the network with its contents being the publicly encrypted Partner_Key.

$$\begin{aligned}&\exists m\text{: Message}(\\ &\quad N'' \text{ Net} = \text{Net} \cup \{m\}\\ &\&\ \text{Type}(m) = \text{Response}\\ &\&\ \text{Source}(m) = \text{Cheater}\\ &\&\ \text{Destination}(m) = \text{Partner}\\ &\&\ \text{Contents}(m) = \text{PEncrypt}(\text{Server\_Public}, \text{Partner\_Key})).\end{aligned}$$

Before invoking the Return_Key transform the add predicate facility of Inatest is used a second time. This time $t2$ is defined as follows:

$$t2 = \text{PEncrypt}(\text{Server\_Public}, \text{Partner\_Key}).$$

The Return_Key transform is invoked next. This transform models the unsuspecting Server processing the partner's response like any other user's response and generating a message for the cheater. The contents of this message is the session key suggested by the partner, which is PDecrypt(Server_Secret, $t2$), encrypted with the key stored in the Server's In_Process table corresponding to the cheater–partner pair, which is PDecrypt(Server_Secret, t). That is,

$$\begin{aligned}&\exists m\text{: Message}(\\ &\quad N'' \text{ Net} = \text{Net} \cup \{m\}\\ &\&\ \text{Type}(m) = \text{Return}\\ &\&\ \text{Source}(m) = \text{Cheater}\\ &\&\ \text{Destination}(m) = \text{Partner}\\ &\&\ \text{Contents}(m) =\\ &\quad \text{Encrypt}(\text{PDecrypt}(\text{Server\_Secret}, t), \text{PDecrypt}(\text{Server\_Secret}, t2))).\end{aligned}$$

Again the add predicate facility is used to define a new symbol for an expression. The symbol $t3$ is defined as follows:

$$t3 = \text{Encrypt}(\text{PDecrypt}(\text{Server\_Secret}, t), \text{PDecrypt}(\text{Server\_Secret}, t2)).$$

Finally, the Compromise_Key transform is invoked with parameter $t3$, and it results in the information

$$\text{Divide}(\text{Decrypt}(\text{Partner\_Key}, t3), \text{Cheater\_Key})$$

becoming part of the intruder's information.

To check whether the expected result ($R1 \in$ Intruder_Info) holds in the resulting state the check result command is executed. This produces the expression

$R1 \in$
{PEncrypt(Server_Public, $R1$),
Times(PEncrypt(Server_Public, Cheater_Key),
PEncrypt(Server_Public, R1)),
PEncrypt(Server_Public, Partner_Key),
Encrypt(PDecrypt(Server_Secret, t), PDecrypt(Server_Secret, $t2$)),
Divide(Decrypt(Partner_Key, $t3$), Cheater_Key),
Server_Public,
Partner_Key}.

To prove this is true it is necessary to use the axioms of the specification. It would be desirable to have this as an automatic feature of the Inatest tool; however, in the current version of the tool it is necessary to apply the axioms manually. That is, the Inatest processor attempts to reduce the expression to true or false, but if it fails it asks the user to act as the theorem prover.

To reduce the expression to true the element

$$\text{Divide}(\text{Decrypt}(\text{Partner\_Key}, t3), \text{Cheater\_Key})$$

is reduced to $R1$.

First the expression represented by $t3$ is reduced. This expression is

$$\text{Encrypt}(\text{PDecrypt}(\text{Server\_Secret}, t), \text{PDecrypt}(\text{Server\_Secret}, t2)),$$

where

$$t = \text{Times}(\text{PEncrypt}(\text{Server\_Public}, \text{Cheater\_Key}), \text{PEncrypt}(\text{Server\_Public}, R1))$$

and

$$t2 = \text{PEncrypt}(\text{Server\_Public}, \text{Partner\_Key}).$$

Using the second axiom t can be reduced to

$$\text{PEncrypt}(\text{Server\_Public}, \text{Times}(\text{Cheater\_Key}, R1)).$$

Then substituting t and $t2$ in $t3$ yields the expression

Encrypt(PDecrypt(Server_Secret,
PEncrypt(Server_Public, Times(Cheater_Key, $R1$))),
PDecrypt(Server_Secret, PEncrypt(Server_Public, Partner_Key))).

Now applying the first axiom to

PDecrypt(Server_Secret, PEncrypt(Server_Public, Times(Cheater_Key, R1)))

yields

Times(Cheater_Key, $R1$).

Also, by using the first axiom

PDecrypt(Server_Secret, PEncrypt(Server_Public, Partner_Key))

reduces to

Partner_Key.

Therefore, $t3$ reduces to

Encrypt(Times(Cheater_Key, $R1$), Partner_Key),

which by applying the fifth axiom is equivalent to

Encrypt(Partner_Key, Times(Cheater_Key, $R1$)).

Substituting this reduced value for $t3$ into the expression of interest yields

Divide(Decrypt(Partner_Key, Encrypt(Partner_Key,
Times(Cheater_Key, $R1$)), Cheater_Key).

By applying the fourth axiom the expression of interest reduces to

Divide(Times(Cheater_Key, $R1$), Cheater_Key),

which by the third axiom reduces to

$R1$.

Therefore, $R1$ is part of the intruder's knowledge. Thus, the Simmons flaw in the TMN protocol has been demonstrated using Inatest.

Executing the Paths command in Inatest at this point generated the following output.

```
#p
Start: request_key(a,b,r1)
No assumptions
Finished
Start: cheater_request(r1)
No assumptions
Finished
Added conditions:
times(pencrypt(server_public,cheater_key),
 pencrypt(server_public,r1)) = t
Start: process_request(cheater,partner,t)
No assumptions
Finished
```

```
Start: partner_response
No assumptions
Finished
Added conditions:
pencrypt(server_public,partner_key) = t2
Start: return_key(cheater,partner,t2)
No assumptions
Finished
Added conditions:
encrypt(pdecrypt(server_secret,t),
 pdecrypt(server_secret,t2)) = t3
Start: compromise_key(t3)
No assumptions
Finished,
```

which is a summary of the penetration scenario and any assumptions made during the execution of the scenario. The penetration scenario is diagrammed in Fig. 8.

4.5. *Discussion*

The intruder actions that are included as transforms in the Ina Jo specification of the TMN protocol are only those needed to model the Simmons attack; all of the possible intruder actions are not included. One of the drawbacks of the Inatest approach is that the analyst determines the intruder actions and the flaw scenarios; there is no disciplined approach nor even a heuristic for determining what actions to include. This *ad hoc* approach to defining user actions and finding flaw scenarios is analogous to software testing; if the appropriate test cases (flaw scenarios) are not tried no flaws will be discovered. However, in other experiments with the approach where the protocol specifications were subjected to formal verification [12], the proof obligations that could not be proved directed the analyst to scenarios that successfully revealed flaws.

One of the motivations for analyzing encryption protocols is that from the flaws that are discovered guidelines for the design of secure encryption protocols may

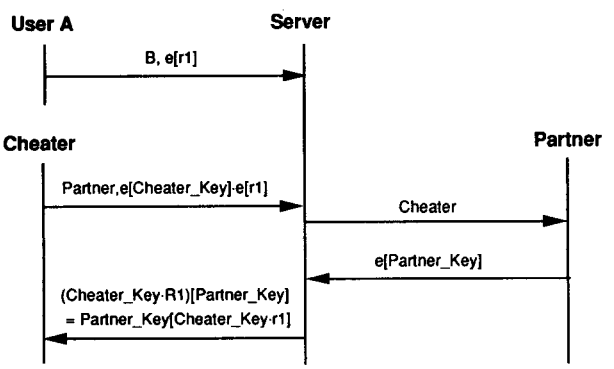

Fig. 8. Simmons attack.

arise. In addition, by expressing the protocol as a formal specification the analyst is provided with a medium that can be studied further.

By using the Inatest system it was possible to demonstrate the Simmons flaw in the TMN protocol. In addition, by further analysis of the specification more general conclusions were drawn. The distribution of the public-key encryption algorithm over multiplication is one of the critical properties of the public-key algorithm that made the Simmons flaw possible. This property is expressed in the second axiom of the formal specification. A generalization that can be made from analyzing the specification is that the flaw will exist for any operation over which the public-key encryption algorithm distributes. That is, Times was used in the formal specification to represent the multiplication operator, but if the intruder had the *foo* and foo^{-1} operators available and the public-key encryption algorithm distributes over *foo*, then the flaw would still exist.

A similar generalization can be made relative to the fifth axiom. That is, in the original TMN protocol example the conventional encryption algorithm used bitwise addition modulo two, which allowed the key and text to be commuted. The flaw would still exist for any conventional algorithm that allowed commutativity of the key and text whenever the text itself was a key.

5. Conclusion

5.1. *Summary*

We have seen how all three protocol analysis tools successfully demonstrated flaws in the TMN protocol. The flaw scenarios revealed by the Interrogator and the NRL Protocol Analysis Tool both required the intruder to capture messages and in some cases to replace them with modified messages. The intruder also had to interject new messages to imitate a normal user's response to a message that was never delivered. Although the flaws in the TMN example were understood ahead of time, it is not hard to see how the tools might be used to help discover unanticipated flaws in other protocols.

The Inatest tool was able to demonstrate the Simmons attack, which enabled the intruder to obtain private data without capturing or altering messages, provided that the intruder and an accomplice were legitimate users. The contribution of the Inatest approach is that it furnishes a means to specify and experiment with a protocol without being constrained by the limitations of a built-in state search or algebraic reduction engine. Neither of the other systems reproduced the arithmetic properties used by Simmons.

5.2. *The Role of Algebra*

It might be asked what kept the Interrogator and the NRL tool from finding the Simmons attack. The answer lies partly in algebraic issues and, in the case of the Interrogator, in the fact that it automates perhaps a bit too much of the search process.

There are actually two separate algebraic features of the Simmons attack. The homomorphic nature of RSA encryption was used to hide the reuse of the first key, $r1$, by the intruder. However, there is a simplified version of the Simmons attack in

which that is not done, but which still defeats the tools. Consider the following message history, in which "+" represents a bitwise modulo-two sum:

$$E \rightarrow S: \mathrm{e}[r1],$$

$$S \rightarrow D: E,$$

$$D \rightarrow S: \mathrm{e}[r3],$$

$$S \rightarrow E: r1 + r3.$$

In the Simmons attack, the parties E and D are the Cheater and Partner, respectively, as shown in Figure 8.

There is no attempt here to avoid the retransmission of $\mathrm{e}[r1]$. However, if the server does not bother to check, the intruder E ends up with $r1 + r3$. Knowing $r3$, $r1$ can be determined.

Now, both the Interrogator and NRL tools specified the protocol with $r1[r2]$ in the last message instead of $r1 + r2$, so the penetration scenario would have $r1[r3]$ instead of $r1 + r3$. The problem is that there is no algebraic rule for encryption that would allow a party who knows $r3$ to obtain $r1$ from $r1[r3]$. It is necessary to understand that a different, symmetric, mechanism was used to combine $r1$ and $r3$. One way to do this with the NRL tool would be to add new rules such as

$$\mathrm{d}(Y, \mathrm{e}(X, Y)) \rightarrow X,$$

$$\mathrm{e}(Y, \mathrm{d}(X, Y)) \rightarrow X.$$

There are presently plans to extend both tools to incorporate modulo-two sum and other symmetric operations in a general fashion. When this is done, it makes sense to see if they can reproduce the simplified attack, as above. Further RSA-specific rules to handle the complete Simmons attack could then be added. Even then, however, it is still not obvious whether the tools would facilitate discovery of the specific Simmons attack.

5.3. *Specification Styles*

The three tools demanded that the protocol be specified in three remarkably different ways. Yet, many of the differences were a matter of syntactic style.

All three tools assumed that each party possessed certain state information. The Interrogator represented the state of a party as a single list, and the significance of the items in the list was implicit in the ordering of the list and the way the items were used to construct messages and state transitions. The NRL tool used separate declarations for each component of the state of a party, and a state component included supplementary information such as a mnemonic variable name and a "round" number as well as the value. The Ina Jo version of a protocol put state information into global matrix variables, but the rows showed the information belonging to each user.

Other differences were conceptual, but not necessarily inherent. An example is the use of "round" numbers in the NRL representation to indicate passage of time. The other specifications did not have anything comparable, but they had enough flexibility so that a similar mechanism might have been provided, if desired.

5.4. *Conclusion*

There were substantial differences in the way the tools were used, reflecting the goals they were designed to pursue. The NRL tool is used primarily to try to prove that a protocol is secure, and discovers flaws as a byproduct of that process. The Interrogator is designed to search for ways of achieving specified insecure states, without guaranteeing that the protocol is secure when the search fails. In return it requires less effort to set up the protocol and conduct the analysis. The Inatest tool is the most flexible, with the ability to specify any algebraic or other properties that may be relevant to the protocol, but the analysis must be controlled manually without the benefit of automatic algebraic reductions.

By working on a common problem and comparing the results obtained by the tools, the authors were able not only to achieve a better understanding of the similarities and contrasts among their approaches, but were stimulated with ideas and motivation to extend and improve the existing tools. It is hoped that the products of this exercise will appear in the not too distant future in the form of powerful and easy-to-use tools for protocol security analysis.

Acknowledgment

We would like to thank Gus Simmons for posing the TMN protocol as a challenge problem to the three of us.

Appendix. Ina Jo TMN Specification

```
SPECIFICATION TMN_Protocol
LEVEL Top_Level

TYPE
   Text,
   Key_Type subtype Text,
   User,
   Message,
   Messages = SET OF Message,
   Message_Type = (Request, Server_Request, Response, Return),
   Information = SET OF Text

CONSTANT
   Type(Message): Message_Type,
   Source(Message): User,
   Destination(Message): User,
   Contents(Message): Text,
   Times(Text, Text): Text,
   Divide(Text, Text): Text,
   Null_Key: Key_Type
```

TYPE
Key = $T''K$: Key_Type($K \neq$ Null_Key),
Keys = SET OF Key

CONSTANT
Server_Public: Key,
Server_Secret: Key,
PEncrypt(Key, Text): Text,
PDecrypt(Key, Text): Text,
Encrypt(Key, Text): Text,
Decrypt(Key, Text): Text,

Cheater: User,
Partner: User,
Cheater_Key: Key,
Partner_Key: Key

AXIOM
$\forall t$: Text(PDecrypt(Server_Secret, PEncrypt(Server_Public, t)) = t)

AXIOM
$\forall t1, t2$: Text, k: Key(
Times(PEncrypt(k, $t1$), PEncrypt(k, $t2$)) = PEncrypt(k, Times($t1$, $t2$)))

AXIOM
$\forall t1, t2$: Text(Divide(Times($t1$, $t2$), $t1$) = $t2$)

AXIOM
$\forall k, t$: Text(Decrypt(k, Encrypt(k, t)) = t)

AXIOM
$\forall k1, k2$: Key(Encrypt($k1$, $k2$) = Encrypt($k2$, $k1$))

VARIABLE
Pending_Key(User, User): Key_Type,
Current_Key(User, User): Key_Type,
In_Process(User, User): Key_Type,
Net: Messages,
Keys_Used: Keys,
Intruder_Info: Information

CRITERION
$\forall k$: Key(($k \in$ Keys_Used & $k \in$ Intruder_Info)
$\rightarrow$ k $\in$ {Server_Public, Partner_Key})

```
INITIAL
    ∀u1, u2: User(
        Current_Key(u1, u2) = Null_Key
      & Pending_Key(u1, u2) = Null_Key
      & In_Process(u1, u2) = Null_Key)
  & Net = EMPTY
  & Keys_Used = {Server_Public, Server_Secret}
  & Intruder_Info = {Server_Public, Partner_Key}

Transform Request_Key(A, B: User, R1: Key)
  Refcond
    R1 ∉ Keys_Used
  Effect
    ∀U1, U2: User(
      N″ Pending_Key(U1, U2) =
        if U1 = A & U2 = B
          then R1
          else Pending_Key(U1, U2)
        fi )
    & N″ Keys_Used = Keys_Used ∪ {R1}
    & ∃m: Message(
        N″ Net = Net ∪ {m}
      & Type(m) = Request
      & Source(m) = A
      & Destination(m) = B
      & Contents(m) = PEncrypt(Server_Public, R1))
    & N″ Intruder_Info = Intruder_Info ∪ {PEncrypt(Server_Public, R1)}

Transform Process_Request(A, B: User, C: Text)
  Refcond
    ∃m: Message(
        m ∈ Net
      & Type(m) = Request
      & Source(m) = A
      & Destination(m) = B
      & Contents(m) = C)
  Effect
    ∀U1, U2: User(
      N″ In_Process(U1, U2) =
          if U1 = A & U2 = B
            then PDecrypt(Server_Secret, C)
            else In_Process(U1, U2)
          fi )
    & ∃m: Message(
        N″ Net = Net ∪ {m}
```

```
        & Type(m) = Server_Request
        & Source(m) = A
        & Destination(m) = B)

  Transform Respond_To_Server(A, B: User, R2: Key)
    Refcond
      R2 ∉ Keys_Used
    Effect
      ∀U1, U2: User(
        N″ Current_Key(U1, U2) =
            if U1 = B & U2 = A
              then R2
              else Current_Key(U1, U2)
            fi )
      & N″ Keys_Used = Keys_Used ∪ {R2}
      & ∃m: Message(
          N″ Net = Net ∪ {m}
        & Type(m) = Response
        & Source(m) = A
        & Destination(m) = B
        & Contents(m) = PEncrypt(Server_Public, R2))
      & N″ Intruder_Info = Intruder_Info ∪ {PEncrypt(Server_Public, R2)}

  Transform Return_Key(A, B: User, C: Text)
    Refcond
      ∃m: Message(
          m ∈ Net
        & Type(m) = Response
        & Source(m) = A
        & Destination(m) = B
        & Contents(m) = C)
      & In_Process(A, B) ≠ Null_Key
    Effect
      ∀U1, U2: User(
        N″ In_Process(U1, U2) =
            if U1 = A & U2 = B
              then Null_Key
              else In_Process(U1, U2)
            fi )
      & ∃m: Message(
          N″ Net = Net ∪ {m}
        & Type(m) = Return
        & Source(m) = A
        & Destination(m) = B
        & Contents(m) = Encrypt(In_Process(A, B),
            PDecrypt(Server_Secret, C))
```

```
    & N″ Intruder_Info = Intruder_Info ∪
      {Encrypt(In_Process(Destination(m), Source(m)),
          PDecrypt(Server_Secret, C))}

Transform Get_Key(A, B: User, C: Text)
  Refcond
    ∃m: Message(
        m ∈ Net
      & Type(m) = Return
      & Source(m) = A
      & Destination(m) = B
      & Contents(m) = C)
    & Pending_Key(A, B) ≠ Null_Key
  Effect
    ∀U1, U2: User(
      N″ Current_Key(U1, U2) =
        if U1 = A & U2 = B
          then Decrypt(Pending_Key(A, B), C)
          else Current_Key(U1, U2)
        fi
      & N″ Pending_Key(U1, U2) =
        if U1 = A & U2 = B
          then Null_Key
          else Pending_Key(U1, U2)
        fi )

Transform Cheater_Request(R1: Key)
  Refcond
    PEncrypt(Server_Public, R1) ∈ Intruder_Info
  Effect
    N″ Keys_Used = Keys_Used ∪
      {PDecrypt(Server_Secret,
        Times(PEncrypt(Server_Public, Cheater_Key),
          PEncrypt(Server_Public, R1)))}
    & ∃m: Message(
        N″ Net = Net ∪ {m}
      & Type(m) = Request
      & Source(m) = Cheater
      & Destination(m) = Partner
      & Contents(m) = Times(PEncrypt(Server_Public, Cheater_Key),
          PEncrypt(Server_Public, R1)))
    & N″ Intruder_Info = Intruder_Info ∪
      {Times(PEncrypt(Server_Public, Cheater_Key),
          PEncrypt(Server_Public, R1))}

Transform Partner_Response
  Refcond
```

```
        ∃m: Message(
              m ∈ Net
            & Type(m) = Server_Request
            & Source(m) = Cheater
            & Destination(m) = Partner)
      Effect
        N″ Keys_Used = Keys_Used ∪ {Partner_Key}
        & ∃m: Message(
              N″ Net = Net ∪ {m}
            & Type(m) = Response
            & Source(m) = Cheater
            & Destination(m) = Partner
            & Contents(m) = PEncrypt(Server_Public, Partner_Key))
        & N″ Intruder_Info = Intruder_Info ∪
            {PEncrypt(Server_Public, Partner_Key)}

  Transform Compromise_Key(C: Text)
    Refcond
      ∃m: Message(
            m ∈ Net
          & Type(m) = Return
          & Source(m) = Cheater
          & Destination(m) = Partner
          & Contents(m) = C)
    Effect
      N″ Intruder_Info = Intruder_Info ∪
        {Divide(Decrypt(Partner_Key, C), Cheater_Key)}

  END Top_Level
  END Simmons_Crypto
```

References

[1] Abadi, M., and M. R. Tuttle, A Semantics for a Logic of Authentication, *Proceedings of the 10th Annual ACM Symposium on Distributed Computing*, ACM Press, New York, August 1991, pp. 201–206.

[2] Bieber, P., A Logic of Communication in a Hostile Environment, *Proceedings of the Computer Security Foundations Workshop III*, IEEE Computer Society Press, New York, June 1990, pp. 14–22.

[3] Boudet, A., Unification in a Combination of Equational Theories: an Efficient Algorithm, *Proceedings of the 10th International Conference on Automated Deduction*, Springer-Verlag, New York, 1990, pp. 292–307.

[4] Burns, J., and C. J. Mitchell, A Security Scheme for Resource Sharing Over a Network, *Comput. Security*, Vol. 9, 1990, pp. 67–76.

[5] Burrows, M., M. Abadi, and R. M. Needham, A Logic of Authentication, *ACM Trans. Comput. Systems*, Vol. 8, No. 1, February 1990, pp. 18–36.

[6] Clocksin, W. F., and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1984.

[7] Denning, D. E., and G. M. Sacco, Timestamps in Key Distribution Protocols, *Comm. ACM*, Vol. 24, No. 8, August 1981, pp. 533–536.

[8] Dolev, D., and A. Yao, On the Security of Public-Key Protocols, *IEEE Trans. Inform. Theory*, Vol. 29, 1983, pp. 198–203.

[9] Eckmann, S. T., and R. A. Kemmerer, INATEST: An Interactive Environment for Testing Formal Specifications, Third Workshop on Formal Verification, Pajaro Dunes, CA, February, 1985, *ACM—Software Engrg Notes*, Vol. 10, No. 4, August 1985, pp. 17–18.

[10] Goldwasser, S., S. Micali, and C. Rackoff, The Knowledge Complexity of Interactive Proof Systems, *Siam J. Comput.*, Vol. 18, No. 1, February 1989, pp. 186–208.

[11] Holzmann, G., *The Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[12] Kemmerer, R. A., Analyzing Encryption Protocols Using Formal Verification Techniques, *IEEE J. Selected Areas Commun.*, Vol. 7, No. 4, May 1989, pp. 448–457.

[13] Kemmerer, R. A., Integrating Formal Methods into the Development Process, *IEEE Software*, September 1990, pp. 37–50.

[14] Kemmerer, R. A., Analyzing the Tatebayashi–Matsuzaki–Newman Protocol Using Inatest, UCSB Report No. TRCS91-05, Computer Science Department, University of California, Santa Barbara, CA, April 1991.

[15] Longley, D., Expert Systems Applied to the Analysis of Key Management Schemes, *Comput. Security*, Vol. 6, 1987, pp. 54–67.

[16] Massey, J. L., Cryptography and Provability, *Proceedings of the Workshop on Mathematical Concepts of Dependable Systems*, Mathematisches Forschunginstitut Oberwolfach, Oberwolfach, April 1990.

[17] Meadows, C., A System for the Specification and Verification of Key Management Protocols, *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, New York, 1991, pp. 182–195.

[18] Meadows, C., Applying Formal Methods to the Analysis of a Key Management Protocol, *J. Comput. Security*, Vol. 1, No. 1, 1992, pp. 5–53.

[19] Merritt, M., and P. Wolper, States of Knowledge in Cryptographic Protocols, unpublished manuscript, 1985.

[20] Millen, J. K., The Interrogator: A Tool for Cryptographic Protocol Security, *Proceedings of the 1984 Symposium on Security and Privacy*, IEEE Computer Society Press, New York, pp. 134–141.

[21] Millen, J. K., S. C. Clark, and S. B. Freedman, The Interrogator: Protocol Security Analysis, *IEEE Trans. Software Engrg.*, Vol. 13, No. 2, February 1987.

[22] Moore, J. H., Protocol Failures in Cryptosystems, *Proc. IEEE*, Vol. 76, No. 5, May 1988, pp. 564–602.

[23] Needham, R. M., and M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Comm. ACM*, Vol. 21, No. 12, December 1978, pp. 993–999.

[24] Rety, P., C. Kirchner, H. Kirchner, and P. Lescanne, NARROWER: A New Algorithm for Unification and Its Applications to Logic Programming, in *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 202, Jean-Pierre Jouannaud, ed., Springer-Verlag, Berlin, 1985, pp. 141–157.

[25] Scheid, J., and S. Holtsberg, Ina Jo Specification Language Reference Manual, TM-6021, Unisys Corporation, Culver City, CA, May 1989.

[26] Simmons, G. J., How To (Selectively) Broadcast a Secret, *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, New York, 1985, pp. 108–113.

[27] Syverson, P. and Meadows, C, A Logical Language for Specifying Cryptographic Protocol Requirements, in *Proceedings of the 1993 IEEE Computer Society Symposium on Security and Privacy*, IEEE Computer Society Press, New York, 1993, pp. 165–177.

[28] Tatebayashi, M., N. Matsuzaki, and D. B. Newman, Key Distribution Protocol for Digital Mobile Communication Systems, in *Advances in Cryptology—CRYPTO '89*, Lecture Notes in Computer Science, Vol. 435, G. Brassard, ed., Springer-Verlag, New York, 1991, pp. 324–333.

[29] Toussaint, M.-J., Separating the Specification and Implementation Phases in Cryptology (Extended Abstract), *Computer Security, ESORICS 92*, Lecture Notes in Computer Science, Vol. 648, Springer-Verlag, Berlin, 1992, pp. 77–101.