Analysis of Security Protocols*

Nancy A. DURGIN John C. MITCHELL
Department of Computer Science
Stanford University
Stanford, CA 94305-9045

Abstract

Several approaches have been developed for analyzing security protocols. Most formal approaches are based on a set of assumptions commonly referred to as the "Dolev-Yao model." In this paper, we use a formalism based on multiset rewriting to describe these modeling assumptions and explain how they are used in protocol analysis. The unifying formalism makes it possible to discuss general optimizations and protocol simplifications that may be used in a variety of approaches. We also summarize some projects using the Dolev-Yao model and explain connections between our formalism and others in current use.

1 Introduction

A variety of protocols based on cryptographic primitives are used to protect access to computer systems and to protect transactions over the internet. Two well-known examples are the Kerberos authentication scheme [21, 20], commonly used to manage access on clusters of interconnected computers, and the Secure Sockets Layer [13], used by internet browsers and servers to carry out secure internet transactions. Handshake or initialization protocols are often used to establish secret session keys for subsequent encrypted communication. These protocols may also be designed to establish or confirm the identity of participants. In many cases, the protocols that are used at the beginning of a transaction or interaction may transmit secret keys across the network. Since these protocols are subtle and critical to the secure operation of the systems they protect, it is important to analyze them in detail before they are deployed.

Over the past decade or two, a variety of methods have been developed for analyzing and reasoning about protocols that use cryptographic primitives. These methods include specialized logics such as BAN logic [7], special-purpose tools designed for cryptographic protocol analysis [18], and theorem proving [37] and model-checking methods using general purpose tools [23, 27, 32, 39, 40]. Many of these approaches are based on a set of assumptions commonly referred to as the "Dolev-Yao model," a reference to an early study [10].

Since there has been no standard presentation of the Dolev-Yao model that illustrates the common assumptions in recent studies, one purpose of this paper is to outline the

*Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and by NSF CCR-9629754.

Dolev-Yao model in a neutral notation. While specialized logics and methods have proven successful in many situations, the focus of this paper is the use of conventional logics and analysis methods, since presentation of protocols within standard formalisms calls our attention directly to the assumptions underlying our protocol models.

In order to make a unified presentation of a set of modeling assumptions, we use a formalism based on multiset rewriting that is designed to have fewer implicit assumptions than other formalisms. Using this notation, we present some basic protocol models, give a precise definition of the intruder capabilities, and describe some basic model variations and optimizations. We also describe some of the connections between our "simplified" notation and theorem-proving and model-checking approaches in current use.

2 An Example

2.1 Needham-Schroeder public key exchange

Some of the basic ideas in protocol analysis can be illustrated using the Needham-Schroeder public key exchange protocol. This protocol was initially presented in [34] as a method for using public-key encryption to "bootstrap" private key cryptography by allowing two parties to securely agree on a private key. We will consider a simplified version of the protocol that omits some initialization steps.

Using a common informal notation, the protocol may be written as follows:

$$\begin{array}{lllll} A & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\ B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\ A & \longrightarrow & B & : & \{N_b\}_{K_b} \end{array}$$

In words, there are two principals in this protocol. It is standard practice to call them Alice (A) and Bob (B). The notation above shows only the communication between them. The internal computation of each principal, which is implicit in the protocol, must be explained in words.

The protocol is run under the standard assumptions of public-key cryptography (see, e.g., [29].) Specifically, we assume there is some encryption and decryption function that both parties have agreed upon in advance. In order to encrypt, a principal must choose an encryption key; decryption is performed using a matching decryption key. A characteristic of public-key cryptography is that a principal may announce an encryption key without revealing information that would allow the matching decryption key to be determined efficiently. Initially, Alice and Bob each have public encryption and private decryption keys. Alice's public encryption key is written K_a and her private decryption key, K_a^{-1}. Similarly, K_b is Bob's public encryption key and K_b^{-1} his private decryption key. Both of the encryption keys are assumed to be known to anyone who wishes to communicate with either one of them. This allows anyone to encrypt a message to Alice or a message to Bob. However, once a message is encrypted with K_a, it is assumed that only Alice can decrypt the message. We use the notation $\{m\}_K$ for the result of encrypting message m with key K. One reason for this post-fix notation is that it highlights the contents of the message, rather than the key or name of the encryption function.

Before the first communication step of the protocol, Alice chooses a number, N_a, at random. This number is called a *nonce*, or "number used once." The significance of choosing this number at random is that a randomly chosen number cannot be guessed or known by any other party with more than an insignificant probability. Even if one of Alice's enemies knows the method for choosing the random number (such as rolling dice or using timing information about keystrokes), Alice uses this method in private and only when she is about to communicate. Therefore, we assume that the number is known only to Alice. To be more precise, we may assume that Alice chooses N_a randomly from among all k-bit numbers. This makes the probability of another party guessing N_a at most $1/2^k$, where Alice may choose k so that this probability is arbitrarily small. However, for the sake of explaining the protocol in simple terms, we will just say that the probability of anyone guessing N_a is negligible.

In the first communication step, Alice sends her name, A, to Bob, along with her nonce, N_a. These are encrypted with Bob's public encryption key so that only Bob can read the message.

When Bob receives the message from Alice, he decrypts the message and learns two things. The first is that Alice would like to communicate with him. We assume that Bob knows the protocol Alice is using. (If this were not the case, the message could be augmented with the name of the protocol.) The second piece of information is Alice's nonce. Bob continues by randomly selecting a nonce, N_b, of his own and sending it to Alice. As he does so, he also sends Alice's nonce back to her. The message containing both nonces is encrypted with Alice's public key so that no one besides Alice can read the message.

When Alice receives the message from Bob, she decrypts it and finds both nonces. When she receives the nonce, N_a, that she sent to Bob, she believes that the message must have come from Bob. The reason for this belief is fairly simple. Alice chose the number in secret in such a way that no one could have randomly guessed it with more than a negligible probability. After choosing N_a, she sent this number out onto the network encrypted with Bob's public key. This made it impossible for anyone other than Bob to determine N_a. Since she has now received a message containing N_a, she can surely believe that Bob read her message and decrypted it. So that Bob can be convinced similarly that Alice received N_b, Alice sends N_b back to Bob, encrypted with Bob's public key.

After three communication steps, it is the intent of this protocol that both Alice and Bob will know that they have communicated with each other. Furthermore, since all communication is sent under encryption, no one else will know the content of either of their messages. Therefore, it seems reasonable to believe that after the protocol is completed, Alice and Bob share two private numbers that cannot be known to any observer.

2.2 Attack on Needham-Schroeder

There is an attack on the Needham-Schroeder public key protocol that was discovered by G. Lowe [22]. The attack involves a malicious third party, whom we will call Mallory. In order to carry out this attack, Mallory must first convince Alice to exchange keys with him. If Alice does so, then Mallory may use the messages from Alice to convince unsuspecting Bob that he is communicating with Alice, even though Alice is not. This allows Mallory to obtain a nonce that was intended for Alice. Since the protocol is intended to allow secret exchange between two parties, it is an error for Mallory to be able to obtain information that Bob intended for Alice.

The attack can be described in the same notation as the protocol. In the following list of communication steps, we see Alice (abbreviated A as before) communicate with Mallory (abbreviated M) and Mallory forward the contents of Alice's message on to Bob (abbreviated B). The steps are described below.

$$
\begin{array}{lcccl}
A & \longrightarrow & M & : & \{A, N_a\}_{K_m} \\
M & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\
B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\
A & \longrightarrow & M & : & \{N_b\}_{K_m} \\
M & \longrightarrow & B & : & \{N_b\}_{K_b}
\end{array}
$$

In the first step, Alice sends her name and nonce to Mallory. This is the standard first step of the protocol. Mallory, however, decrypts the message using his private decryption key, encrypts it with Bob's public key, and forwards the message on to Bob. When Bob receives this message, it appears to him that Alice is trying to initiate the protocol with him.

Bob responds by generating a nonce N_b and sending it back, along with the nonce N_a that he received in the previous message. These are encrypted with Alice's public key and sent to Alice. Alice has no idea that this message is coming from Bob. Since the message has the same form as if it had come from Mallory, Alice responds according to the third step of the protocol. Therefore Alice sends the second nonce (which was actually generated by Bob, not Mallory) to Mallory. This message is encrypted with Mallory's public key, allowing Mallory to decrypt the message and learn the nonce N_b generated by Bob. In the final step, Mallory sends Bob the nonce N_b that he's expecting, thus convincing Bob that he has successfully completed the protocol with Alice.

One question that often comes up in discussing this attack is whether it is realistic to assume that a malicious attacker, Mallory, could convince an honest party, Alice, to start a conversation with him. Here is one kind of situation where something like this might occur. We can think of Alice as a bank customer and Bob as the branch where she has an account. If Mallory opens up a dishonest bank in order to deceive customers, then he might advertise better interest rates than Bob on the internet. Alice notices Mallory's advertisement and decides to open a bank account with Mallory by initiating the key-exchange protocol. Mallory may then try to use Alice's message to secretly gain access to her account with Bob. This would involve an attack of the form described here.

2.3 Improving Needham-Schroeder

The attack on the Needham-Schroeder public-key protocol can be prevented by a very simple addition to the protocol [23]. Specifically, if Bob includes his identity in the second message of the protocol, then the "Mallory-in-the-middle" attack does not work. The repaired protocol is:

$$
\begin{array}{lcccl}
A & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\
B & \longrightarrow & A & : & \{B, N_a, N_b\}_{K_a} \\
A & \longrightarrow & B & : & \{N_b\}_{K_b}
\end{array}
$$

We can see how this works by looking at the first few steps of the attack.

$$
\begin{array}{lcccl}
A & \longrightarrow & M & : & \{A, N_a\}_{K_m} \\
M & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\
B & \longrightarrow & A & : & \{B, N_a, N_b\}_{K_a}
\end{array}
$$

As before, Alice sends her name and nonce to Mallory. Mallory decrypts the message and encrypts it with Bob's public key, forwarding the result on to Bob. When Bob receives this message, it again appears that Alice is trying to initiate the protocol with him.

In the repaired protocol, however, Bob responds sending a nonce N_b, the nonce N_a that he received in the previous message, and his name, B. These are encrypted with Alice's public key and sent to Alice. Unlike the original protocol, Alice can immediately see that this message is coming from Bob, not Mallory. She can then realize that something is wrong and abort the protocol. Specifically, she sent her first message out to Mallory, but receives a message back indicating that Bob is responding to her message. Since she did not initiate the protocol with Bob, Alice is able to detect the attack by Mallory.

This argument shows that changing the protocol eliminates one attack, but how do we know that there are no attacks on the repaired protocol? There are several analyses of the repaired Needham-Schroeder protocol that establish correctness under a specific set of assumptions [23, 27, 36, 41].

3 Methods for protocol analysis

3.1 Security analysis

Security analysis involves analysis of system activities when a malicious attacker interferes with the operation of the system. From a scientific point of view, the most significant problem in analyzing security protocols is accurate modeling of the malicious adversary or *intruder*. However, there is some important interaction between the model of the system and the intruder model. Specifically, in order to model one form of intrusion, it is necessary to model the system in sufficient detail to allow this intrusion to take place. For example, if we are going to analyze the possibility of guessing passwords from a file of encrypted passwords, then we must include the fact that passwords are composed of a sequence of letters and numbers. On the other hand, in analyzing a protocol to make sure that a password is always supplied, it may be adequate to represent a password by some kind of indivisible token or atomic value. Generally speaking, the more detailed model, the more difficult the analysis and the more useful the results. Since analyzing systems for security properties can be extremely difficult, it is also generally desirable to use the simplest model that is likely to shed some light on the problem, at least to start off with.

Historically, there have been many methods for analyzing security protocols. In addition to informal arguments, there have been several basic varieties of mathematical arguments.

Turing-machine models One interesting approach is the oracle Turing machine model proposed by Bellare and Rogaway and used in several studies [3, 4, 42]. In this model, a protocol is represented by a series of oracles, one for each step of each principal of the protocol. The protocol is analyzed by considering the computations that a probabilistic polynomial-time Turing machine can carry out, given access to all of these oracles. This is a detailed form of protocol analysis, using essentially the same assumptions about cryptographic primitives as in modern complexity-based cryptography. However, it can be difficult to carry out a proof of protocol correctness under these assumptions.

Special-purpose logics Beginning with the so-called BAN logic of Burroughs, Abadi, and Needham, several special-purpose logics have been developed for protocol analysis [7, 16]. In BAN logic, a protocol is represented by a set of formulas expressing properties of the steps of the protocol. Formalizing a protocol in this way can require creativity and insight, since this form of logical presentation of a protocol expresses not only the computation of the protocol participants, but the purpose or goals of each step. Although BAN and related logics have proven useful for analyzing many protocols, they also have their shortcomings. In particular, the flawed Needham-Schroeder protocol can be proved correct in BAN logic [7, 23]. This is not because BAN logic is unsound. (The inference rules of BAN logic are in fact correct for some semantics [2].) The problem lies in the way that the Needham-Schroeder protocol is formalized in the logic. In a sense, the logic seems too "coarse" to express a subtle property that allows Lowe's attack to succeed.

Methods based on protocol traces The class of methods that we will discuss in this paper are based on some form of execution of the steps of a protocol, possibly interleaved with steps of a malicious adversary. The approaches we will consider all use essentially the same basic assumptions about network communication and the capabilities of the adversary. These assumptions, largely derived from [10] and ideas expressed in [34], provides a reasonable balance between tractability and insight into the behavior of protocols. However, since many reasonable attacks are not modeled within this common framework, it is likely that at least one branch of future research will focus on more detailed intruder models.

3.2 "Dolev-Yao" intruder capabilities

Although [10] analyzes a restricted set of protocols in which each action depends only on the message received, not the state of the participant involved in the action, we will refer to a general set of modeling assumptions with wider applicability as the "Dolev-Yao" model. In the Dolev-Yao-based model, the protocol adversary has the following capabilities:

- Read any message and block further transmission
- Decompose a message into parts and remember them
- Generate fresh data as needed
- Compose a new message from known data and send

The messages generated by the intruder may be composed of any information supplied to the intruder initially (such as public keys of protocol participants) and data obtained by overhearing or intercepting messages. By combining a read with resend, we can easily obtain the effect of passively reading a message without preventing another party from also receiving it.

In this model, the intruder is not allowed to choose random messages. The reason is that the formal settings that use this model do not provide any effective way of taking probability into consideration. As a result, several kinds of realistic attacks are outside the scope of this model.

3.3 Explicit-intruder methods

The Dolev-Yao model can be used as the basis for a variety of protocol analysis techniques. The approaches we are concerned with in this paper may be called "explicit-intruder methods", since they involve an explicit model of the protocol and an explicit model of the protocol intruder. A representative way of using an explicit-intruder method to analyze a protocol would involve the following sequence of steps:

1. *Formulate the protocol.* If the protocol is described in detail, then this generally involves simplifying the protocol by identifying the key steps and primitives. On the other hand, formal analysis requires a more detailed description of the behavior of protocol participants than the high-level descriptions sometime seen in the literature. For example, the common notation

$$\begin{array}{lcccl} A & \rightarrow & B & : & \{n\}_K \\ B & \rightarrow & A & : & \{n+1\}_K \end{array}$$

   specifies that Alice send a number n to Bob, under encryption by key K, and Bob returns $n+1$ encrypted in the same way. However, this notation does not clearly specify what Bob will do in response to a message other than $\{n\}_K$. In order to analyze the effects of a malicious adversary, we need to understand the way that each participant will respond to any possible message.

2. *Add an adversary to the system.* An adversary of the form described above may be represented by a process that maintains a set of "known" data and chooses nondeterministically between possible actions at each step. If the protocol is designed to be secure against various partial failures, these must be included. For example, if the protocol is intended to preserve secrecy even when previously-used keys are made public, then these keys should be made known to the adversary at the appropriate point in the execution of the protocol.

3. *State the desired correctness condition.* Typical correctness conditions are related to authentication and secrecy. For example, if a server S enters a state in which it commits to open a session with client C, then C must have previously requested a session with S. We may also specify that each datum that is initially secret remains secret. In addition to secrecy, a general class of correctness properties are called *correspondence properties*[44]. A correspondence property has the form, "if principal B receives (or states, etc.) x at some point, then principal A must have previously sent (or received or chosen, etc.) x."

4. *Analyze the protocol.* This may be done by trying to prove that for all values of the system parameters (number of sessions, etc.), the protocol is correct, or using some form of search tool to search for executions of the protocol and intruder in which the intruder causes some correctness condition to be violated.

5. *Experiment with alternate formulations and repeat.* In examples which are found to be incorrect, it can be informative to repair the detected errors, either by strengthening the protocol or redirecting the efforts of the adversary. In cases where a protocol appears correct, it may be useful to investigate the consequences of strengthening the adversary, possibly by revealing some of the "secret" information, to see how robust the protocol is to partial breaches of security. In analysis methods that use a finite bound on the number of participants or size of the memory of various principals, it may be useful to try to increase these bounds on repeated runs.

Many researchers have used this basic approach to successfully study a variety of protocols. We examine these approaches in more detail in Section 5.

4 Multiset rewriting with existential quantification

4.1 Protocol Notation

We introduce a formalism for describing a class of infinite-state systems. The formalism is similar in many respects to standard rewrite systems [19, 31], with two main differences. The first is that instead of representing information by a single expression, we use multisets of first-order atomic formulas. (A multiset is similar to a set, but with counting of duplicates.) The second main difference is that the formalism has a basic mechanism for choosing "new" symbols. This is important for modeling protocols that choose a new nonce or generate encryption keys.

The syntax involves terms, facts and rules. If we want to represent a system in this formalism, we begin by choosing a vocabulary, or *first-order signature*. This is a standard notion from multi-sorted first-order logic [11].

Signatures A first-order signature consist of a set of sorts, together with function symbols and predicate symbols with specific sorts. The sorts indicate the kinds of data that will be used in the model. For example, the sorts used in a protocol may be key, msg, nonce for encryption keys, message contents and nonces. Function symbols are names for functions on the sorts of the signature. For example, an encryption function might have sort

$$\mathsf{encrypt} : \mathsf{key} \times \mathsf{msg} \to \mathsf{cipher}$$

where cipher is the sort for ciphertexts (encrypted text). In any signature, each function symbol must have a fixed set of parameter sorts (one for each function argument) and a result sort. A function with no arguments is called a *constant symbol*. Finally, a multi-sorted first-order signature has a set of predicate symbols, each with a fixed set of parameter sorts.

An illustrative of a first-order signature may be derived from the definition of classical cryptosystem from [43]:

> A *classical cryptosystem* is a five-tuple $\langle \mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D} \rangle$ satisfying the following conditions:
>
> 1. $\mathcal{P}$ is a finite set of possible plaintexts
> 2. $\mathcal{C}$ is a finite set of possible ciphertexts
> 3. $\mathcal{K}$, the keyspace, is a finite set of possible keys
> 4. For each $K \in \mathcal{K}$, there is an encryption function $e_K : \mathcal{P} \to \mathcal{C}$ in $\mathcal{E}$ and a corresponding decryption function $d_K : \mathcal{C} \to \mathcal{P}$ in $\mathcal{D}$ such that $d_K(e_K(x)) = x$ for every plaintext $x \in \mathcal{P}$.

If we change notation slightly, so that e_K is considered as a function $\mathsf{encrypt}(\_, K)$ depending on the choice of key, then we can associate a first-order signature with a classical cryptosystem. Specifically, we use three sorts plain, cipher, key and functions $\mathsf{encrypt} : \mathsf{plain} \times \mathsf{key} \to \mathsf{cipher}$ and $\mathsf{decrypt} : \mathsf{cipher} \times \mathsf{key} \to \mathsf{plain}$.

The relation between encryption and decryption can be stated using the equation

$$\mathsf{decrypt}(\mathsf{encrypt}(x, k), k) = x$$

for all $k : \mathcal{K}$ and $x : \mathcal{P}$. While it is possible to add equational axioms to the formalism described in this paper, we will not do so. Instead, we will avoid the extra complication of equational axioms and use predicates instead. For example, if we add predicate symbols Know_k, Know_c and Know_p to identify the keys, ciphertexts and plaintexts known to some principal, then we could express the property

If $\mathsf{Know}_\mathsf{k}(k)$ and $\mathsf{Know}_\mathsf{c}(\mathsf{encrypt}(x, k))$ then $\mathsf{Know}_\mathsf{p}(x)$.

The reason for three separate predicates, Know_k, Know_c and Know_p, is that each predicate can only be applied to arguments of a fixed sort. Since keys, ciphertext and plaintext are represented using three separate sorts, we need three predicates. The ”if-then’ reasoning described here can be formalized using rules, defined below.

Terms The terms over a signature are the set of expressions formed by applying functions to arguments. In each case, a function must be applied to arguments of the correct sort. For example, if $f : s \to t$ and $x : t$, then $f(x)$ is a well-formed term since the argument sort of f matches the sort of x. As suggested be this example, terms may contain variables, but each variable must have an associated sort. A variable is not allowed to be used with different sorts in different expressions associated with the same system. (All of this can be formalized using an inductive definition of the well-formed terms and their sorts, but we assume that most readers will be familiar with these standard concepts from logic.)

Facts A *fact* is a first-order atomic formula. This means that a fact is the result of applying a predicate symbol to terms of the correct sorts.

States In our formalism, a *state* is a multiset of facts (all over the same signature).

Rules State transitions are written using two multisets of facts, in the following syntactic form:

$$F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$$

The meaning of this rule is that if some state S contains facts $F_1, \ldots F_k$, then one possible next state is the state S' that is similar to S, but with:

- facts $F_1, \ldots F_k$ removed,
- $G_1, \ldots G_m$ added, where $x_1 \ldots x_m$ are replaced by new symbols.

If there are free variables in the rule $F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$, these are treated as universally quantified. In an application of a rule, these free variables may be replaced by any terms.

For example, consider the state

$$S = \{P(f(a)), P(b)\}$$

and rule

$$P(x) \longrightarrow P(f(x)).$$

Then one possible next state is obtained by instantiating the rule to

$$P(f(a)) \longrightarrow P(f(f(a))).$$

With this substitution, we can remove $P(f(a))$ from S and obtain the next state $S' = \{P(f(f(a))), P(b)\}$. We can then use a different instance of the rule,

$$P(b) \longrightarrow P(f(b))$$

to reach state $S'' = \{P(f(f(a))), P(f(b))\}$. It is also possible to reach S'' from S by performing these replacements in the opposite order.

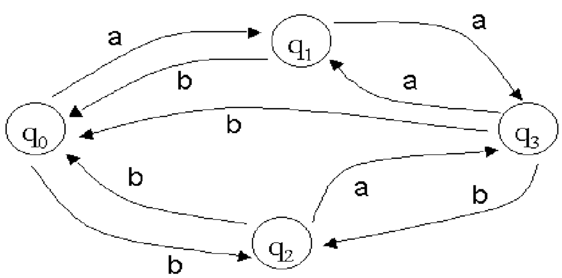
If a function is invertible, then this can also be expressed as a rule. For example, the rule

$$P(f(x)) \longrightarrow Q(x)$$

involves recovering the data x from $f(x)$. We will use rules of this form to describe decryption of encrypted messages.

4.2 Finite-State Example

As a first example, without existential quantification, we can describe the following finite-state automaton.



In addition to naming the states, we will also formalize the input and resulting transitions. The vocabulary for this example will have three sorts, st for automaton states, symb for input symbols and and string for lists of input symbols. The predicates, functions, constants and rules are as follows.

- Predicates: state : st, input : string
- Function: cons : symb $\times$ string $\rightarrow$ string
- Constants: q_0, q_1, q_2, q_3 : st, a, b : symb, and nil : string.

For simplicity, we will write $a \cdot x$ for cons(a, x).

There are nine rules, one for each state transition. Since all of these have essentially the same form, we just list the rules for transitions from states q_0 and q_1.

$$\mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{a} \cdot \mathsf{x}) \longrightarrow \mathsf{State}(\mathsf{q_1}), \mathsf{Input}(\mathsf{x})$$

$$\mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{b} \cdot \mathsf{x}) \longrightarrow \mathsf{State}(\mathsf{q_2}), \mathsf{Input}(\mathsf{x})$$

$$\mathsf{State}(\mathsf{q_1}), \mathsf{Input}(\mathsf{a} \cdot \mathsf{x}) \longrightarrow \mathsf{State}(\mathsf{q_3}), \mathsf{Input}(\mathsf{x})$$

$$\mathsf{State}(\mathsf{q_1}), \mathsf{Input}(\mathsf{b} \cdot \mathsf{x}) \longrightarrow \mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{x})$$

A sample derivation gives us automata state transitions from q_0 to q_1 and back on input $a \cdot b \cdot \mathsf{nil}$. We will write this out as a sequence of states, starting with the multiset $\{\mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{nil})\}$ that represents the automaton in state q_0 with input string $a \cdot b \cdot \mathsf{nil}$.

$$\{\mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{nil})\} \quad \longrightarrow \quad \{\mathsf{State}(\mathsf{q_1}), \mathsf{Input}(\mathsf{b} \cdot \mathsf{nil})\} \quad \longrightarrow \quad \{\mathsf{State}(\mathsf{q_0}), \mathsf{Input}(\mathsf{nil})\}$$

It should be easy to see that if we begin with a system state consisting of one fact about the state of the automaton, and one fact about the input string, then we can only reach other system states of this form. In particular, we can never reach a system state where the automaton is in two states.

4.3 Existential Quantification

It is possible to give finite descriptions of infinite-state systems using function symbols. For example, if we have $0 : \mathsf{nat}$ and $\mathsf{suc} : \mathsf{nat} \rightarrow \mathsf{nat}$, then we can write expressions for arbitrarily many natural numbers. If each system state has a natural number, then we will have infinitely many possible system states.

Existential quantification provides an alternate way of describing systems with infinitely many possible states. The value of existential quantification, for security protocols, is that it provides a direct mechanism for choosing a new value that is different from other values used in the execution of a system. Since many protocols involve choosing fresh nonces, fresh encryption keys, and so on, existential quantification seems like a useful primitive for describing security protocols.

The way that existential quantification is used in our formalism is based on the existential elimination rule from natural deduction. This proof rule is commonly written as follows.

$$(\exists\ \mathbf{elim}) \qquad \frac{\exists x.\phi \qquad \begin{matrix}[y/x]\phi \\ \vdots \\ \psi\end{matrix}}{\psi} \qquad \begin{array}{l} y \text{ not free in any other} \\ \text{hypothesis in the proof of } \psi \end{array}$$

If we have an existentially quantified axiom, $\exists x.\phi$, then this rule says that if we wish to prove some formula ψ, we can choose a new symbol y for the "x that is presumed to exist" and proceed to derive ψ from $[y/x]\phi$. The side condition "y not free in any other hypothesis in the proof of ψ" means that the only hypothesis in the proof of ψ that can contain y is the hypothesis $[y/x]\phi$.

For those not familiar with natural deduction, an explanation may help. Consider the following fact about the natural numbers.

$$\forall x.\, \exists y.\, (y > x) \wedge \mathsf{prime}(y)$$

This formula says that for every natural number x, there is a prime number greater than x. Suppose we want to use this to prove that for every number z, there is a product of two distinct primes greater than z. In informal reasoning, we might say, "Let y be a prime greater than z." In this initial step, we have chosen a new name, y, for a prime greater than z. We can then go on to chose another number greater than y if we want a product of two distinct primes greater than z. In each step, we are using this natural deduction proof rule to choose new names for existentially quantified variables. Additional information on natural deduction can be found in [38, 15], for example.

Turing Machine Example We can see how existential quantification allows us to describe infinite-state systems by axiomatizing a Turing machine. Let us assume we have some specific Turing machine M. We choose a vocabulary for describing states of the machine and its input, and we write rules to describe transitions according to machine state and input. The rules will depend on the specific machine M, but the basic method can be applied to any Turing machine.

The first predicate in this example is used to describe the current machine state and tape position. The other two predicates describe the contents of a tape cell and the order (adjacency) between cells.

$\mathsf{Curr}(\mathsf{state}, \mathsf{cell})$	current state, tape pos.
$\mathsf{Cont}(\mathsf{cell}, \mathsf{symbol})$	contents of cell is symbol
$\mathsf{Adj}(\mathsf{cell}, \mathsf{cell})$	keep cells in order

We also need names for the states of the machine M, names for the cells at the beginning and end of the tape, and names for the symbols that may appear on the tape (0,1, and blank). The reason we have an end-of-tape cell, c_{eot}, is that we will represent an unbounded tape by including rules that will allow us to allocate as many tape cells as needed. In other words, we will represent the Turing machine tape by explicitly constructing the finite list of cells that the machine has looked at, one at a time.

$q_0, q_1, q_2, \ldots$	finite set of states
$c_0, c_1, \ldots, c_{eot}$	initial tape cells
$0, 1, \not{b}$	tape symbols

There are three classes of transition rules: tape maintenance rules, transition rules that correspond to Turing machine moves that move the head to the right, and transition rules that correspond to moving the head left. Initially, we will start the machine with two tape cells, the leftmost cell c_0 and the rightmost end-of-tape cell c_{eot}. This is expressed by the fact

$$\mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_{\mathsf{eot}})$$

At any step in the computation, we can apply the tape maintenance rule

$$\mathsf{Adj}(\mathsf{c}, \mathsf{c}_{\mathsf{eot}}) \longrightarrow \exists c'.\mathsf{Adj}(\mathsf{c}, \mathsf{c}'), \mathsf{Cont}(\mathsf{c}', \not{b}), \mathsf{Adj}(\mathsf{c}', \mathsf{c}_{\mathsf{eot}})$$

Informally, this rule "says" that if cell c is adjacent to the end-of-tape cell, then we can allocate a new cell c' and place c' between c and the end-of-tape cell. The new cell will contain a blank. An example computation below shows how this rule can be used.

The rules for the actual moves of the Turing machine will depend on the structure of the specific machine M we wish to represent. Suppose that Turing machine M moves to the right, if it is in state q_i with symbol 0 on the tape cell currently under the tape head. If the move of M, in this case, is to state q_j, writing 1 into the tape cell, we will have a rule of the following form:

$$\mathsf{Curr}(\mathsf{q_i}, \mathsf{c}), \mathsf{Cont}(\mathsf{c}, 0), \mathsf{Adj}(\mathsf{c}, \mathsf{c}') \longrightarrow \mathsf{Curr}(\mathsf{q_j}, \mathsf{c}'), \mathsf{Cont}(\mathsf{c}, 1), \mathsf{Adj}(\mathsf{c}, \mathsf{c}')$$

If, instead of moving right, the machine would move left in this case, we would instead have the transition rule

$$\mathsf{Curr}(\mathsf{q_i}, \mathsf{c}), \mathsf{Cont}(\mathsf{c}, 0), \mathsf{Adj}(\mathsf{c}', \mathsf{c}) \longrightarrow \mathsf{Curr}(\mathsf{q_j}, \mathsf{c}'), \mathsf{Cont}(\mathsf{c}, 1), \mathsf{Adj}(\mathsf{c}, \mathsf{c}')$$

Note that moving to the right, we assume that there is a tape cell to the right of the tape head. This assumption can be satisfied by using the tape maintenance rule if needed before executing this Turing machine move.

Sample computation Although the exact moves will depend on the exact Turing machine that is represented by this method, we can illustrate the use of existential quantification by showing some example moves of a sample machine. Let us consider a Turing machine that reads the input tape until two consecutive **0**'s are read, then inserts a **1** after the both of them and moves to the left over the second **0**. Suppose that the machine starts in state q_0 and remains in this state until it reaches a cell containing 1. At that point, the machine changes to state q_1 to "remember that it has seen a **0**" and moves right. Then the first few moves of the machine on input **100** might appear as follows:

$$\begin{array}{l}
\mathsf{Curr}(\mathsf{q}_0, \mathsf{c}_0), \mathsf{Cont}(\mathsf{c}_0, 1), \mathsf{Cont}(\mathsf{c}_1, 0), \mathsf{Cont}(\mathsf{c}_2, 0), \mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_1), \mathsf{Adj}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{Adj}(\mathsf{c}_2, \mathsf{c}_{\mathsf{eot}}) \\
\longrightarrow \mathsf{Curr}(\mathsf{q}_0, \mathsf{c}_1), \mathsf{Cont}(\mathsf{c}_0, 1), \mathsf{Cont}(\mathsf{c}_1, 0), \mathsf{Cont}(\mathsf{c}_2, 0), \mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_1), \mathsf{Adj}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{Adj}(\mathsf{c}_2, \mathsf{c}_{\mathsf{eot}}) \\
\longrightarrow \mathsf{Curr}(\mathsf{q}_1, \mathsf{c}_2), \mathsf{Cont}(\mathsf{c}_0, 1), \mathsf{Cont}(\mathsf{c}_1, 0), \mathsf{Cont}(\mathsf{c}_2, 0), \mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_1), \mathsf{Adj}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{Adj}(\mathsf{c}_2, \mathsf{c}_{\mathsf{eot}})
\end{array}$$

At this point, the appropriate transition will be a move to the right on to the next cell, where the machine will write a **1**. However, this would place the tape head over the special "end-of-tape" marker. Since we would like the machine to proceed as if the tape were infinite, we must use the tape maintenance rule

$$\mathsf{Adj}(\mathsf{c}, \mathsf{c}_{\mathsf{eot}}) \longrightarrow \exists c'.\mathsf{Adj}(\mathsf{c}, \mathsf{c}'), \mathsf{Cont}(\mathsf{c}', \not{b}), \mathsf{Adj}(\mathsf{c}', \mathsf{c}_{\mathsf{eot}})$$

to insert a new cell in front of the end-of-tape cell. This gives us the transition

$$\begin{array}{l}
\mathsf{Curr}(\mathsf{q}_1, \mathsf{c}_2), \mathsf{Cont}(\mathsf{c}_0, 1), \mathsf{Cont}(\mathsf{c}_1, 0), \mathsf{Cont}(\mathsf{c}_2, 0), \mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_1), \mathsf{Adj}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{Adj}(\mathsf{c}_2, \mathsf{c}_{\mathsf{eot}}) \\
\longrightarrow \mathsf{Curr}(\mathsf{q}_1, \mathsf{c}_2), \mathsf{Cont}(\mathsf{c}_0, 1), \mathsf{Cont}(\mathsf{c}_1, 0), \mathsf{Cont}(\mathsf{c}_2, 0), \mathsf{Cont}(\mathsf{c}_3, \not{b}), \\
\quad \mathsf{Adj}(\mathsf{c}_0, \mathsf{c}_1), \mathsf{Adj}(\mathsf{c}_1, \mathsf{c}_2), \mathsf{Adj}(\mathsf{c}_2, \mathsf{c}_3), \mathsf{Adj}(\mathsf{c}_3, \mathsf{c}_{\mathsf{eot}})
\end{array}$$

which inserts a new blank cell, c_3 in front of the end-of-tape marker. Then the machine head may be moved right over the new blank square and write a **1**. Although it would be possible to apply a transition rule moving the machine head over the end-of-tape marker, the end-of-tape marker has no contents. Since each machine move requires a tape cell with some contents (possibly including the blank $\not{b}$), a derivation that places the Turing machine head over the end-of-tape marker will "hang" the machine and have no effect on the set of accepting computations.

4.4 Simplified Needham-Schroeder

The formal presentation of the Needham-Schroeder protocol can be decomposed into several meaningful parts. We will begin with a simplified form of the protocol that does not use any encryption, then discuss public-key encryption and the intruder model separately.

Without encryption, the Needham-Schroeder protocol proceeds as follows:

$$\begin{array}{ccccl}
A & \longrightarrow & B & : & N_a \\
B & \longrightarrow & A & : & N_a, N_b \\
A & \longrightarrow & B & : & N_b
\end{array}$$

We can describe this protocol in our notation using the predicates A_i, B_i, N_i for $0 \leq i \leq 3$, with the following intuitive meaning:

$A_i(\ \dots\ )$	Alice is in state i, with indicated data
$B_i(\ \dots\ )$	Bob is in state i, with indicated data
$N_i(\ \dots\ )$	The network contains message i, with indicated data

The data associated with the state of some principal, or a network message, will depend on the particular state or message. Each principal begins in local state 0, with no data. Therefore, predicates A_0 and B_0 are predicates with no arguments. When Alice chooses a nonce, she moves into local state 1. Therefore, predicate A_1 is a predicate of one argument, intended to be the nonce chosen by Alice. Similarly, predicate B_1 has two arguments, the data received from Alice in message one of the protocol and the nonce chosen by Bob for his response.

Using this predicates, we can state the protocol using four transition rules:

$$\begin{array}{rcl} A_0() & \longrightarrow & \exists x.A_1(x), N_1(x) \\ B_0(), N_1(x) & \longrightarrow & \exists y.B_1(x,y), N_2(x,y) \\ A_1(x), N_2(x,y) & \longrightarrow & A_2(x,y), N_3(y) \\ B_1(x,y), N_3(y) & \longrightarrow & B_2(x,y) \end{array}$$

Each rule corresponds to an action by a principal. In the first rule, Alice chooses a nonce, sends it on the network, and remembers the nonce by moving into a local state that retains the nonce value. In the second step, Bob receives a message on the network, chooses his own nonce, transmits it and saves it in his local state. In the third step, Alice receives Bob's message and replies, while in the fourth step Bob receives Alice's final message.

Here is a sample trace generated from these rules, beginning from state A_0, B_0. Spacing is used to separate the facts that participate in each step from those that do not.

$$\begin{array}{rrcll} B_0(), & A_0() & \longrightarrow & A_1(\mathsf{nA}), N_1(\mathsf{nA}), & B_0() \\ A_1(\mathsf{nA}), & B_0, N_1(\mathsf{nA}) & \longrightarrow & B_1(\mathsf{nA},\mathsf{nB}), N_2(\mathsf{nA},\mathsf{nB}), & A_1(\mathsf{nA}) \\ B_1(\mathsf{nA},\mathsf{nB}), & A_1(\mathsf{nA}), N_2(\mathsf{nA},\mathsf{nB}) & \longrightarrow & A_2(\mathsf{nA},\mathsf{nB}), N_3(\mathsf{nB}), & B_1(\mathsf{nA},\mathsf{nB}) \\ A_2(\mathsf{nA},\mathsf{nB}), & B_1(\mathsf{nA},\mathsf{nB}), N_3(\mathsf{nB}) & \longrightarrow & B_2(\mathsf{nA},\mathsf{nB}), & A_2(\mathsf{nA},\mathsf{nB}) \end{array}$$

4.5 Formalizing the intruder

There are two main parts of the Dolev-Yao model as commonly used in protocol analysis. The first is the set of possible intruder actions, applied nondeterministically throughout execution of the protocol. The second is a "black-box" model of encryption and decryption. We explain the intruder actions here, with the encryption model presented in Section 4.6.

As described in Section 3.2, the Dolev-Yao intruder may nondeterministically choose among the following actions at each step:

- Read any message and block further transmission
- Decompose a message into parts and remember them
- Generate fresh data as needed
- Compose a new message from known data and send

In our formalization of this model, the intruder processes data in three phases. The first is to read and decompose data into parts. The second is to remember parts of messages, and the third is to compose a message from parts it remembers. We illustrate the basic form of the intruder actions using one unary network-message predicate N_1 and one binary network-message predicate N_2. Using predicates D for decomposable messages and M for the intruder "memory", the basic rules for intercepting and decomposing messages are

$$\begin{array}{lcllcl} N_1(x) & \longrightarrow & D(x) & \quad D(x) & \longrightarrow & M(x) \\ N_2(x,y) & \longrightarrow & D(x,y) & \quad D(x,y) & \longrightarrow & M(x), M(y) \end{array}$$

While the predicate D may appear to be an unnecessary intermediary here, a protocol with more complicated messages will lead to more interesting ways of destructuring messages. As noted in [26], it is important in proof search to separate the decomposition phase from the composition phase, which is accomplished here using separate D and C predicates. The rules for composing messages from parts are written using the C, for "composable", predicate as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl} M(x) & \longrightarrow & C(x) & C(x) & \longrightarrow & N_1(x) \\ C(x), C(y) & \longrightarrow & C(x,y) & C(x,y) & \longrightarrow & N_2(x,y) \end{array}$$

The rule for generating new data is

$$\longrightarrow \ \exists x.M(x)$$

The reason we need the last transition rule (which can be applied any time without any hypothesis) is that the intruder may need to choose new data in order to trick an honest participant in a protocol. This is illustrated in the following attack on the simplified (obviously insecure) form of the Needham-Schroeder protocol.

For the simplified example at hand, we can compose rules to eliminate the D and C predicates as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl} N_1(x) & \longrightarrow & M(x) & M(x) & \longrightarrow & N_1(x), M(x) \\ N_2(x,y) & \longrightarrow & M(x), M(y) & M(x), M(y) & \longrightarrow & N_2(x,y), M(x), M(y) \\ N_3(x) & \longrightarrow & M(x) & M(x) & \longrightarrow & N_3(x), M(x) \end{array}$$

This reduces the number of steps in the following trace, which has actions of the honest participants in the left column and actions of the intruder indented. For simplicity, duplicate copies of $M(\,)$ facts are not shown, since these have no effect on the execution of the protocol or intruder.

$B_0(), A_0()$	Initial configuration
$\longrightarrow A_1(\mathsf{nA}), N_1(\mathsf{nA}), B_0()$	Alice chooses nonce and sends
$\quad\longrightarrow A_1(\mathsf{nA}), B_0(), M(\mathsf{nA})$	Intruder intercepts message nA
$\quad\longrightarrow A_1(\mathsf{nA}), B_0(), M(\mathsf{nA}), M(n)$	Intruder generates fresh value n
$\quad\longrightarrow A_1(\mathsf{nA}), N_1(n), B_0(), M(\mathsf{nA}), M(n)$	Intruder sends n to Bob
$\longrightarrow B_1(n, \mathsf{nB}), N_2(n, \mathsf{nB}), A_1(\mathsf{nA}), M(\mathsf{nA}), M(n)$	Bob receives, generates nonce, replies
$\quad\longrightarrow A_1(\mathsf{nA}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Intruder intercepts message with n
$\quad\longrightarrow A_1(\mathsf{nA}), N_2(\mathsf{nA}, \mathsf{nB}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Intruder sends nA
$\longrightarrow A_2(\mathsf{nA}, \mathsf{nB}), N_3(\mathsf{nB}), B_1(n, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Alice responds
$\longrightarrow B_2(n, \mathsf{nB}), A_2(\mathsf{nA}, \mathsf{nB}), M(\mathsf{nA}), M(n), M(\mathsf{nB})$	Bob changes to final state, indicating successful completion of protocol

In this attack, the intruder intercepts messages between A and B, replacing data so that the two principals have a different view of the messages that have been exchanged. Specifically, the intruder replaces Alice's nonce nA by a value n chosen by the intruder. When Bob responds to the altered message, the intruder intercepts the result and replaces n by nA so that Alice receives the message she expects.

4.6 Modeling Perfect Encryption

The commonly used "black-box" model of encryption may be written in our multiset notation using the following vocabulary. For concreteness, we discuss public-key encryption. Symmetric or private-key encryption can be characterized similarly. We assume that plaintexts have sort plain and ciphertexts have sort cipher.

The formal presentation of public-key cryptography uses the following symbols.

- Additional sorts: $\mathsf{e\_key}, \mathsf{d\_key}$
- Predicate: $\mathsf{KeyPair}(\mathsf{e\_key}, \mathsf{d\_key})$
- Function: $\mathsf{enc} : \mathsf{e\_key} \times \mathsf{plain} \rightarrow \mathsf{cipher}$

We could also include a decryption function $\mathsf{dec} : \mathsf{d\_key} \times \mathsf{cipher} \rightarrow \mathsf{plain}$. However, it seems simpler to write protocols using pattern-matching (encryption on the left-hand-side of a rule) to express decryption.

Example The core Needham-Schroeder protocol with encryption begins with each principal generating and publishing a key pair. Therefore the "local state 0" predicate for each principal will contain a key pair. For example, Bob's initial actions can be stated as the following rules:

$$\begin{array}{rcl} & \longrightarrow & \exists k : \mathsf{e\_key}.\, \exists k' : \mathsf{d\_key}.\, B_0(k),\ \mathsf{KeyPair}(k, k') \\ B_0(k) & \longrightarrow & \mathsf{Announce}(k), B_0(k) \end{array}$$

The first rule (without hypotheses) lets Bob generate a key pair, while the second announces the public key so that other principals can choose to communicate with Bob. Alice similarly chooses and publishes her public key. After doing so, she chooses a principal to communicate with from the set of announced public keys and transmits a message.

$$A_0(k), \mathsf{Announce}(k') \quad \longrightarrow \quad \exists x.\ A_1(k, k', x), N_1(\mathsf{enc}(k', \langle x, k\rangle)), \mathsf{Announce}(k')$$

The following transition rule then allows Bob to decrypt the message from Alice.

$$B_0(k), N_1(\mathsf{enc}(k, \langle x, k'\rangle)) \quad \longrightarrow \quad \exists y.\ B_1(k, k', x, y), N_2(\mathsf{enc}(k', \langle x, y\rangle))$$

A complete presentation with slightly more general initialization steps is given in Appendix A, for the interested reader.

Intruder To model the encryption capabilities of the intruder, we add a decomposition and a composition rule to the intruder model. The decomposition rule allows the intruder to decrypt a message (or part of a message) when the decryption key is known.

$$D(\mathsf{enc}(k, x)),\ \mathsf{KeyPair}(k, k'),\ M(k') \longrightarrow D(x),\ \mathsf{KeyPair}(k, k'),\ M(k')$$

The composition rule allows the intruder to encrypt a message with any encryption key known to the intruder.

$$M(k),\ C(x) \longrightarrow C(\mathsf{enc}(k, x)),\ M(k)$$

An important point about our formulation of the Dolev-Yao intruder is that this is the entire definition. In particular, every derivation according to the protocol rules and the intruder rules corresponds to a protocol execution in which the principals act as intended and the intruder behaves according to the actions allowed in the Dolev-Yao model. There is no need to impose additional side conditions on which data the intruder looks at, or freshness of generated values. The restrictions on intruder actions are represented explicitly in the rules, and the freshness of generated values is guaranteed by the way existential quantification is realized as a transition rule.

Some basic properties of this approach to encryption are:

- Encryption and decryption can be performed only if the appropriate key is known to the intruder.
- The intruder can produce $\mathsf{enc}(key, msg)$ if key and msg are known to the intruder.

The first assumption is often realistic, since for most encryption functions, we need the plaintext and key to compute the cipher text and, conversely, the ciphertext and key to compute the plaintext by decryption. In practice, there is some probability of guessing the keys or text at random, but these probabilities are usually small (or can be made small by choosing an appropriate key length).

The second property above merits some discussion. In our formal model, it is easy to identify the rules that allow the intruder to generate an encrypted message. Specifically, if the intruder generates $N_i(\mathsf{enc}(k, x))$ then it must be by one of these rules:

$$\begin{array}{lll} N_i(\mathsf{enc}(k,x)) \longrightarrow M(enc(k,x)) & & \text{(replay)} \\ M(k), M(x) \longrightarrow M(enc(k,x)) & & \text{(knows parts)} \end{array}$$

In words, either the intruder has already heard the encrypted message on the network, or the intruder has learned the plaintext and the key. Unfortunately, these rules do not reflect the properties of certain specific encryption functions, such as RSA [29]. Specifically, RSA encryption is calculated using modular arithmetic by

$$\mathsf{enc}(k, msg) = \text{msg}^k \bmod N$$

As a result, it is easy to see that RSA encryption satisfies the following relation

$$\mathsf{enc}(k, x * y) = \mathsf{enc}(k, x) * \mathsf{enc}(k, y)$$

In words, the encryption of a message $x * y$ can be computed easily from the encryption of x and the encryption of y, without knowing x or y. If we want to include this property of RSA in analysis of some protocol, we can do so by including this rule, or some variant with similar effect:

$$N_i(\mathsf{enc}(k, x)) \longrightarrow M(\dots)\dots \longrightarrow N_i(\mathsf{enc}(k, c * x))$$

Unfortunately, in finite-state analysis, including multiplication on our formalization of a protocol leads to infinitely many possible states. However, bounded versions of this rule have been used successfully to analyze protocols based on RSA [32].

5 Explicit-intruder methods based on the Dolev-Yao model

5.1 Summary of existing methods

Previous work on protocol analysis using the Dolev-Yao model breaks down into two basic types – those that attempt to formally prove correctness properties for a protocol, and those that perform an analysis to find attacks on the protocol. In both cases, as noted in Section 3.3, the protocol and correctness conditions are first formalized into a notation appropriate to the method, and then the analysis is performed on the formalized version of the protocol (i.e. perhaps with certain assumptions and simplifications necessary to make the analysis feasible).

The approaches that attempt to prove protocol correctness tend to formulate the problem without placing any artificial bounds on the number of participants or the messages space. Paulson uses the "Inductive method" [37] to analyze the set of messages that can be produced by the protocol participants and prove properties about all possible traces of a protocol. Abadi and Gordon express security guarantees as equivalences between processes in spi calculus, proving that the protocol is equivalent to some ideal protocol [1]. Bolignano uses automatic abstraction methods [5, 6] to prove properties of protocols. Fabrega, Herzog and Guttman use Strand spaces [12] to prove correspondency properties about participants in a protocol.

Another approach is to use symbolic search, with pruning. An error state is identified, and then search proceeds backwards to find a possible protocol execution that could lead to the error. Meadows uses a symbolic representation of the protocol states along with the NRL analyzer to perform a hybrid analysis that uses symbolic backward proof search combined with using forward invariants to prove that certain states are inaccessible [28]. Millen used a similar symbolic proof search approach with his Interrogator [18].

Several attacks on common protocols have been discovered by using exhaustive finite state analysis. These methods approximate the infinite state system by a finite one, and then search the space for error conditions. Lowe, Roscoe and Schneider have all used FDR, a model-checker based on the CSP process algebra [23, 39, 25, 24, 40]. Mitchell et al. used the Murφ model-checker to analysis the SSL protocol, by writing an implementation of SSL in a specialized input language [32, 33]. Clarke et al. use a state search combined with an axiomatic intruder model to examine a variety of protocols [26, 9].

We summarize some of these approaches in more detail in the following sections, with the goal of illustrating connections between our multiset rewrite notation and other formalisms.

5.2 Paulson's Inductive Method

Paulson has developed a proof method for security protocols that uses induction over protocol traces, where a trace is a list of events that occur in some run of a protocol [37]. Although in principle the method could be carried out using pencil and paper, Paulson and his group have used the method to develop machine-checked proofs using the Isabelle HOL system.

Paulson's method uses agents, messages and events. These categories can be described using BNF notation:

$$\begin{array}{rcl} agent & ::= & server \mid friend_i \mid spy \\ msg & ::= & agent \mid number \mid nonce \mid key \mid \langle msg, msg \rangle \mid hash(msg) \mid crypt\ key\ msg \\ event & ::= & says\ agent\ agent\ msg \mid notes\ agent\ msg \end{array}$$

In words, an agent may be the server (a distinguished agent in some client–server protocols), a friend or honest agent, or the spy. A message may be any of the kinds of data that appear in the protocol under consideration, such as the name of an agent, a number, a nonce, a key, a pair of messages, and so on. An event is either a network transmission, in which some agent says a message to another, or some event having to do with the internal state of some agent. The "internal state" events are called $notes$ and involve a message since messages are the general form of data in a protocol.

A $trace$ is list of $events$. Using ML notation [35], we write $::$ for infix cons operation on list. In words, the empty list nil is a trace and, if evs is a trace and e is an event, then $e :: evs$ is a trace.

Paulson inductively defines operators on sets of messages such as

$$\begin{array}{lcl} parts & : & msg\ set \rightarrow msg\ set \\ analz & : & msg\ set \rightarrow msg\ set \\ synth & : & msg\ set \rightarrow msg\ set \end{array}$$

In words, $parts$ gives the set of all parts of message from the given set, $analz$ the set of all parts of message accessible to the intruder, and $synth$ the set of messages that could be composed from messages in the given set. These operators are used in the definition of the set of traces of a protocol.

For a given protocol P, the final step in formalizing the protocol and intruder is to inductively define a set P of traces of the protocol. The clauses of this inductive definition look similar to the rules we would use to define the protocol in our multiset rewriting notation. There is always the condition

$$nil \in P$$

saying that the empty trace is a trace of the protocol. An example clause for the intruder might be

$$(evs \in P) \wedge (B \neq spy) \wedge (X \in synth(analz(spies\ evs))) \Rightarrow (says\ spy\ B\ X) :: evs \in P$$

This rule lets the spy transmit a message composed of parts of messages accessible to the spy. While Paulson has carefully defined meaningful predicates that make a rule like this seem natural and readable, this is actually a complicated statement since it relies on the inductive definitions of predicates $synth$ and $analz$. An example clause for an honest agent action in a protocol might be

$$\begin{array}{l} (evs \in P) \wedge (A \neq B) \wedge (B \neq server) \wedge (nonce\ na \notin used(evs)) \\ \Rightarrow (says\ A\ server\ (init\ A\ B\ na)) :: evs \in P \end{array}$$

In words, this rule states that at any point in the execution of a protocol, an agent A may initiate another run of the protocol with another agent B. The preconditions that $A \neq B$ and $B \neq server$ are included to avoid meaningless runs of the protocol. Note that in this approach, an inductively defined operator $used$ is needed to state that a nonce is "fresh" and has not appeared previously in the trace.

It is possible to write each of the clauses of the inductive definitions of common protocols in our notation as rules $F_1, \ldots, F_k \longrightarrow \exists x_1 \ldots \exists x_j . G_1, \ldots, G_n$. For example, the above rule can be written as:

$$Agent(A), Agent(B) \longrightarrow \exists x. N_0(A, server, \langle init, A, B, x \rangle), Agent(A), Agent(B)$$

Note that the use of existentials avoids the need to specify side-conditions on the freshness of the nonce value, and we've omitted the conditions on A and B being different.

The final, and most important, step in Paulson's method it to prove properties about the set P of traces of a protocol. Some example theorems might be that it is possible for the protocol to run to termination, and that it is not susceptible to certain attacks. The reason for proving that the protocol can complete, i.e., there exist traces in which the intruder does not attack the protocol, is to develop some faith in the formalization of the protocol. An example security condition is that no secret key ends up in the set $analz(spies\ evs)$ of facts known by the spy unless an agent explicitly reveals the secret key to the spy.

5.3 Process Calculus

There are a variety of process calculus notations described in the literature. We will consider the spi-calculus of Abadi and Gordon [1], because it includes both the ν operator from pi-calculus and cryptographic primitives. We will show how a protocol written in our notation can be translated into spi-calculus. Since our notation does not include any specification or proof method, we will not discuss the specification or proof methods associated with spi-calculus.

There is a straightforward translation between the rewrite notation and process calculus, where each rule in the rewrite notation becomes a process, predicates are replaced by port names and pattern-matching is replaced by explicit desctructuring. In order to represent the existential operator, we use the ν operator from pi-calculus.

For example, when translating a simple rule from rewrite notation to a spi-calculus process,

$$B_1(x) \longrightarrow N_2(x), B_2(x)$$

becomes:

$$b_1\langle x\rangle.\,(\overline{n_2}\langle x\rangle \mid \overline{b_2}\langle x\rangle)$$

In words, a rule requiring fact $B_1(x)$ becomes a process that reads x on channel b_1. Following the same correspondence between predicates and channels, a rule that produces facts $N_2(x)$ and $B_2(x)$ becomes a process that outputs x on channels n_2 and b_2.

For a more complex rule, involving existentials, pairs, and multiple predicates on the left hand side,

$$B_1(x, y), N_1(x) \longrightarrow \exists z.N_2(x, z), B_2(x, y, z)$$

becomes:

$$\lceil b_1(p).n_1(x)\rceil.let\ (x, y) = p \text{ in } (\nu z)(\overline{n_2}\langle(x, z)\rangle \mid \overline{b_2}\langle(x, y, z)\rangle)$$

Where (x, y) is a standard shorthand for the pairing x,y, and we introduce the notation $\lceil\Box\rceil$ to indicate two messages that must both be available simultaneously.

As an example, we consider the "Wide Mouthed Frog Protocol" protocol studied by Gordon and Abadi:

$$\begin{array}{llcll}
1. & A & \rightarrow & S & : & \{K_{AB}\}_{K_{AS}} \\
2. & S & \rightarrow & B & : & \{K_{AB}\}_{K_{SB}} \\
3. & A & \rightarrow & B & : & \{M\}_{K_{AB}}
\end{array}$$

In this protocol, the principals A and B share keys K_{AS} and K_{SB} with the server S. A creates a new key K_{AB} and sends it to the server encrypted under K_{AS}. The server sends the key to B encrypted under K_{SB}, and now B can use the key to communicate securely with A (i.e. decrypt the secret M from A).

Note that we will discuss a simplified protocol which only considers the two participants A and B, each allowed to participate in only one session. The more general case of an arbitrary number of participants and an arbitrary number of sessions can also be represented in both Spi Calculus and the rewrite notation, but we will not consider it here.

We can represent this protocol in our rewrite notation as follows:

Role of A:

$$
\begin{array}{lcll}
A_1 & : & A_0(ks,m) & \rightarrow \quad \exists x.A_1(ks,m,x), N_1(enc(ks,x)) \\
A_2 & : & A_1(ks,m,x) & \rightarrow \quad A_2(ks,m,x), N_3(enc(x,m))
\end{array}
$$

Role of B:

$$
\begin{array}{lcll}
B_1 & : & B_0(ks), N_2(enc(ks,x)) & \rightarrow \quad B_1(ks,x) \\
B_2 & : & B_1(ks,x), N_3(enc(x,m)) & \rightarrow \quad B_2(ks,x,m)
\end{array}
$$

Role of S:

$$
S_1 \quad : \quad N_1(enc(ks,x)), S_0(ks,k's) \quad \rightarrow \quad N_2(enc(k's,x))
$$

Initial State:

$$
K = \{A_0(K_{AS}, M), B_0(K_{BS}), S_0(K_{AS}, K_{BS})\}
$$

We can translate this into spi calculus as follows:

$$
\begin{array}{rcl}
A_0(M) & \triangleq & \overline{a_0}\langle (K_{AS}, M)\rangle \\
A_1 & \triangleq & a_0\langle (ks,m)\rangle.(\nu K_{AB})(\overline{n_1}\langle \{K_{AB}\}_{ks}\rangle.\overline{a_1}\langle (ks,m,K_{AB})\rangle) \\
A_2 & \triangleq & a_1\langle (ks,m,k)\rangle.\overline{n_3}\langle \{m\}_k\rangle \\
S_0 & \triangleq & \overline{s_0}\langle (K_{AS}, K_{AB})\rangle \\
S_1 & \triangleq & \lceil s_0\langle (ks,k's)\rangle.n_1(x)\rceil.case\ x\ of\ \{y\}_{ks} in\ \overline{n_2}\langle \{y\}_{k's}\rangle \\
B_0 & \triangleq & \overline{b_0}\langle K_{SB}\rangle \\
B_1 & \triangleq & \lceil b_0\langle ks\rangle.n_2(x)\rceil.case\ x\ of\ \{y\}_{ks} in\ \overline{b_1}\langle y\rangle \\
B_2 & \triangleq & \lceil b_1\langle y\rangle.n_3(z)\rceil.case\ z\ of\ \{w\}_y in\ F(w) \\
Inst(M) & \triangleq & (\nu K_{AS})(\nu K_{SB})(A_0(M) \mid !A_1 \mid !A_2 \mid S_0 \mid !S_1 \mid B_0 \mid !B_1 \mid !B_2)
\end{array}
$$

Most of this translation is straightforward, but note the following:

1) We must use the $\lceil \Box \rceil$ notation to ensure that a process doesn't consume part of a message, unless all required parts are available. This is because in the rewrite notation, the rule will not fire unless all predicates of the left hand side are available, but in process calculus, normally each message is consumed sequentially. This would result in deadlock situations, without the additional "simultaneous messages" feature.

2) Note that a rule in the rewrite notation can be used as many times as needed. This is the equivalent of the ! notation in spi calculus, which represents an unlimited number of processes. The initial knowledge in this example (the initial state of the 3 participants, A_0, B_0, and S_0) is a predicate that can be used only once. This is represented in spi calculus by a single copy of a process which produces a message representing the state.

Abadi and Gordon express the Wide Mouthed Frog Protocol in spi calculus as follows:

$$
\begin{array}{rcl}
A(M) & \triangleq & (\nu K_{AB})(\overline{c_{AS}}\langle \{K_{AB}\}_{K_{AS}}\rangle.\overline{c_{AB}}\langle \{M\}_{K_{AS}}\rangle) \\
S & \triangleq & c_{AS}(x).case\ x\ of\ \{y\}_{K_{AS}} in\ \overline{c_{SB}}\langle \{y\}_{K_{SB}}\rangle \\
B & \triangleq & c_{SB}(x).case\ x\ of\ \{y\}_{K_{SB}} in\ c_{AB}(z).case\ z\ of\ \{w\}_y in\ F(w) \\
Inst(M) & \triangleq & (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B)
\end{array}
$$

Note that this is similar to the version we derive from the rewrite notation, with the difference being that the role of each participant is collapsed into a single, more complicated process.

5.4 Finite-state methods

Since finite-state methods are limited to a finite search space, it is necessary to limit the potentially infinite behaviors of the protocol and intruder.

In the protocol analysis model under consideration, there are two main sources of infinite behaviors: a possibly unlimited number of protocol participants, each participating in multiple simultaneous protocol runs, and messages of possibly unlimited size (i.e. from unlimited nesting of operations such as encryption and data concatenation). In order to perform a finite state analysis on this infinite system, some finite approximations must be made.

In our rewrite notation, we can make these finite approximations in a straightforward way. To limit the number of participants and the number of roles for each participant, we can simply limit the number of times rules with the existential operator can be applied. In practice, the initialization steps of the protocol can be performed in advance of the simulation, yielding a set of initial predicates for each role (i.e. if only one $A_0()$ and one $B_0()$ are provided, and the rules that generate new copies aren't included, then we effectively limit the simulation to one instance of the A role, and one instance of the B role).

The size of terms can be limited by restricting the application of repeated functions, of the form $f(f(f(f(x))))$. In practice, this can be accomplished by placing type restrictions on the rules. For example, if only doubly-nested encryption is allowed, then the intruder's rules would explicitly provide functions of type:

$$
\begin{aligned}
\mathsf{enc}_1 \quad &: \quad \mathsf{e\_key} \times \mathsf{plain} \longrightarrow \mathsf{cipher}_1 \\
\mathsf{enc}_2 \quad &: \quad \mathsf{e\_key} \times \mathsf{cipher}_1 \longrightarrow \mathsf{cipher}_2
\end{aligned}
$$

Thus preventing the applying an encryption operation to data of type cipher_2.

As mentioned in Section 5.1, the FDR system has been successfully used to analyze a variety of protocols [23, 39, 25, 24, 40]. FDR is a finite-state system based on process calculus, so it is straightforward to translate the FDR notation into the rewrite notation, using similar techniques to those used for the spi calculus notation, plus the finite state approximations mentioned above.

Other finite state systems such as Murφ [32] and the system used by Clarke [26, 9] use custom notations, though correspondences with protocols expressed in rewrite notation should be straightforward in principle.

6 Conclusion

Many methods for analyzing security protocols are based on a set of assumptions commonly referred to as the "Dolev-Yao model," a reference to an early study [10]. The logic-based formalism described in this paper, using existential quantification to choose new values, provides a useful meta-system for examining the Dolev-Yao model. Using this formalism, we can characterize protocols by formulating a rewrite rule (or several related rules) for each step of each protocol participant. An advantage of our formalism, in comparison with some other notations or systems, is that the executions of the protocol and intruder correspond precisely to sequences of rewrites. In contrast, some other methods involve defining a set of possible traces, and then imposing complicated side-conditions to distinguish correct traces from incorrect ones.

Although we have not described many examples in this paper, it is possible to develop standard theories (sets of rewrite rules) for common protocol modules. For example, we have written small theories for initialization steps such as generating and announcing public keys. By building up a library of such theories, we may develop a modular approach to security protocol analysis. Since our multiset rewriting notation is a fragment of linear logic [14], it is also possible to do mechanized reasoning directly from our system, using linear-logic tools such as *LLF* [8], *Forum* [30] or *Lygon* [17].

Acknowledgments: Thanks to Grit Denker, Patrick Lincoln, Jon Millen, Mark Mitchell, Andre Scedrov, Vitaly Shmatikov and Ulrich Stern for helpful discussions.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997. Revised and expanded versions to appear in *Information and Computation* and as SRC Research Report 149 (January 1998).

[2] M. Abadi and M. Tuttle. A semantics for a logic of authentication. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, pages 201–216, 1991.

[3] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - CRYPTO '93, Lecture Notes in Computer Science, Vol. 773*, 1994.

[4] M. Bellare and P. Rogaway. Provably secure session key distribution - the three party case. In *Proc. 27th ACM Symposium on the Theory of Computing*, 1995.

[5] D. Bolignano. An approach for the formal verification of cryptographic protocols. In *ACM Conference on Computer and Communications Security*, pages 106–118. ACM Press, 1996.

[6] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *Computer Aided Verification. 9th International Conference*, pages 131–142, 1997.

[7] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, 1989. Also appeared as SRC Research Report 39 and, in a shortened form, in ACM Transactions on Computer Systems 8, 1 (February 1990), 18-36.

[8] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96*, pages 264–275, New Brunswick, NJ, July 1996. IEEE Computer Society Press.

[9] E.M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[10] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[11] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[12] F. Fabrega, J. Herzog, and J. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. IEEE Symposium on Security and Privacy*, pages 160–171, 1998.

[13] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. `draft-ietf-tls-ssl-version3-00.txt`, November 18 1996.

[14] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[15] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[16] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. Technical report, Univ. Cambridge Computing Laboratory, 1990.

[17] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, pages 391–405, Munich, Germany, July 1996. Springer-Verlag LNCS 1101.

[18] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

[19] J.W. Klop. Term rewriting: a tutorial. *EATCS Bulletin*, 32:143–182, 1987.

[20] J.T. Kohl and B.C. Neuman. The Kerberos network authentication service (version 5). Internet Request For Comment RFC-1510, September 1993.

[21] J.T. Kohl, B.C. Neuman, and T.Y. Ts'o. *The evolution of the Kerberos authentication service*, pages 78–94. IEEE Computer Society Press, 1994.

[22] G. Lowe. An attack on the Needham-Schroeder public-key protocol. *Info. Proc. Letters*, 56:131–133, 1995.

[23] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[24] G. Lowe. SPLICE/AS: A case study in using CSP to detect errors in security protocols. Technical report, Programming Reserach Group, Oxford, 1996.

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

[26] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols. *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[27] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: a comparison of two approaches. In *Proc. European Symposium On Research In Computer Security*. Springer Verlag, 1996.

[28] C. Meadows. The NRL protocol analyzer: an overview. *J. Logic Programming*, 26(2):113–131, 1996.

[29] A.J. Menzies, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[30] Dale Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[31] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[32] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proc. IEEE Symp. Security and Privacy*, pages 141–151, 1997.

[33] J.C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of ssl 3.0. In *Seventh USENIX Security Symposium*, pages 201–216, 1998.

[34] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[35] L.C. Paulson. *ML for the working programmer*. Cambridge Univ. Press, 1991.

[36] L.C. Paulson. Mechanized proofs of security protocols: Needham-schroeder with public keys. Technical Report 413, University of Cambridge Computer Lab, 1997.

[37] L.C. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83, 1997.

[38] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.

[39] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Soc Press, 1995.

[40] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[41] S. Schneider. Using CSP for protocol analysis: the Needham-Schroeder public-key protocol. Technical Report CSD-TR-96-14, Royal Holloway, University of London, 1996.

[42] V. Shoup and A. Rubin. Session key distribution using smart cards. In *Eurocrypt '96*, pages 321–331, 1996.

[43] D.R. Stinson. *Cryptography: theory and practice*. CRC Press, 1995.

[44] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *Proc. IEEE Symp. on Research in Security and Privacy*, 1993.

A Example: Needham-Schroeder Public Key Protocol

For the interested reader, we give the full theory of the three-step core of the Needham-Schroeder public-key protocol.

$$\begin{array}{lcccl} A & \longrightarrow & B & : & \{A, N_a\}_{K_b} \\ B & \longrightarrow & A & : & \{N_a, N_b\}_{K_a} \\ A & \longrightarrow & B & : & \{N_b\}_{K_b} \end{array}$$

A.0.1 Initialization Theory

(Unlimited number of principals)

GOODGUY: $\rightarrow \exists k_e.k_d.GoodGuy(k_e, k_d), KeyPair(k_e, k_d)$

(Unlimited number of bad (compromised) keys)

BADKEY: $\rightarrow \exists k_e.k_d.KeyPair(k_e, k_d), BadKey(k_e, k_d)$

(Unlimited number of roles)

ROLA: $GoodGuy(k_e, k_d) \rightarrow GoodGuy(k_e, k_d), A_0(k_e)$

ROLB: $GoodGuy(k_e, k_d) \rightarrow GoodGuy(k_e, k_d), B_0(k_e)$

(Announce keys (for public key since we want to omit server))

ANNK: $GoodGuy(k_e, k_d) \rightarrow AnnKey(k_e), GoodGuy(k_e, k_d)$

ANNKB: $BadKey(k_e, k_d) \rightarrow AnnKey(k_e), BadKey(k_e, k_d)$

A.0.2 Protocol Theory

Protocol Theory $\mathcal{A}$:

A1: $AnnKey(k'_e), A_0(k_e) \quad \to$
$\exists x.A_1(k_e, k'_e, x), N_{S1}(enc(k'_e, \langle x, k_e \rangle)), AnnKey(k'_e)$

A2: $A_1(k_e, k'_e, x), N_{R2}(enc(k_e, \langle x, y \rangle)) \quad \to$
$A_2(k_e, k'_e, x, y), N_{S3}(enc(k'_e, y))$

Protocol Theory $\mathcal{B}$:

B1: $B_0(k_e), N_{R1}(enc(k_e, \langle x, k'_e \rangle)), AnnKey(k'_e) \quad \to$
$\exists y.B_1(k_e, k'_e, x, y), N_{S2}(enc(k'_e, \langle x, y \rangle)), AnnKey(k'_e)$

B2: $B_1(k_e, k'_e, x, y), N_{R3}(enc(k_e, y)) \quad \to \quad B_2(k_e, k'_e, x, y)$

A.0.3 Intruder Theory

(Send/Receive message – Intruder receives decomposable messages, and sends composable messages)

REC: $N_{Si}(x) \quad \to \quad D(x)$

SND: $C(x) \quad \to \quad N_{Ri}(x)$

(Composition and Decompostion)

COMP: $C(x), C(y) \quad \to \quad C(\langle x, y \rangle)$

DCMP: $D(\langle x, y \rangle) \quad \to \quad D(x), D(y)$

(Learning/using information)

LRN: $D(x) \quad \to \quad M(x)$

USE: $M(x) \quad \to \quad C(x), M(x)$

(Encryption and Decryption)

ENC: $M(k_e), C(x) \quad \to \quad C(enc(k_e, x)), M(k_e)$

DEC: $M(k_d), KeyPair(k_e, k_d), D(enc(k_e, x)) \quad \to$
$M(k_d), KeyPair(k_e, k_d), D(x), M(enc(k_e, x))$

(Generate atomic knowledge)

GEN: $\quad \to \quad \exists x.M(x)$

(Intruder learns announced keys)

LRNK: $AnnKey(k_e) \quad \to \quad M(k_e), AnnKey(k_e)$

(Intruder learns bad keys)

LRNKB: $BadKey(k_e, k_d) \quad \to \quad M(k_e), M(k_d), BadKey(k_e, k_d)$