AUTOMATING SECURITY PROTOCOL ANALYSIS

THESIS

Stephen W. Mancini, 1Lt, USAF

AFIT/GCS/ENG/04-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/04-12

AUTOMATING SECURITY PROTOCOL ANALYSIS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Stephen W. Mancini, BS

1Lt, USAF

March 2004

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AFIT/GCS/ENG/04-12

AUTOMATING SECURITY PROTOCOL ANALYSIS

Stephen W. Mancini, BS

1Lt, USAF

Approved:

____/signed/_____	_9 Mar 2004_
Major Robert P. Graham, Jr., PhD (Chairman)	Date
____/signed/_____	_9 Mar 2004_
Richard A. Raines, Ph.D., AFIT (Member)	Date
____/signed/_____	_9 Mar 2004_
Henry B. Potoczny, Ph.D., AFIT (Member)	Date

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Major Robert Graham. There is no way this thesis would have been done this well without his guidance. Also, I would like to thank the members of my committee, Dr. Richard Raines and Dr. Henry Potoczny, for their support and assistance. Finally, I would like to acknowledge Dr. Sylvan Pinsky, my sponsor from NSA. The trips to Baltimore and California proved very valuable in my understanding of the whole Strand Space Theory.

Stephen W. Mancini

Table of Contents

Page

Acknowledgments iv
Table of Contents v
List of Figures viii
List of Tables x
Abstract xi
I. Introduction 1
1.1 Background 1
1.2 Problem Statement 1
1.3 Summary of Results 2
1.4 Thesis Overview 2
II. Literature Review 4
2.1 Chapter Overview 4
2.2 Symmetric/Asymmetric Cryptography 5
2.3 Strand Spaces and Security Protocols 6
2.3.1 Strand Spaces – Brief Overview 7
2.3.2 Understanding the Penetrator 9
2.3.3 Authentication Tests 10
2.3.4 Tests Applied to Needham-Schroeder 14
2.4 Security Protocol Examples 15
2.4.1 Types of Protocols 15
2.4.2 Needham-Schroeder Protocol 16
2.4.3 Penetration of Needham-Schroeder Defined 17

2.4.4 Kerberos 18

2.5 Related Work in the Automation of Security Protocol Verification 20

2.5.1 Failure Divergences Refinement Checker.....21

2.5.2 Process Calculus.....21

2.5.3 Athena.....22

2.5.4 Security Modeling in Maude.....22

2.5.5 Multi-Set Rewriting.....24

2.6 Summary.....24

III. Methodology 25

3.1 Chapter Overview.....25

3.2 Problem Review 25

3.3 Software Overview.....26

3.4 Java Based Protocol Analysis.....26

3.4.1 Layout of Java Program.....26

3.4.2 Description of Parsing Operation 28

3.4.3 Description of Analysis Operation 29

3.5 Specific Protocols analyzed using Java.....32

3.5.1 Verifying Presence of Tests.....33

3.5.2 Analyzing Needham-Schroeder.....36

3.6 Summary.....39

IV. Analysis and Results.....40

4.1 Chapter Overview.....40

4.2 Results of Protocol Analyses.....40

4.2.1 Wide Mouth Frog Protocol.....41

4.2.2 Yahalom Protocol.....42

4.2.3 Woo-Lam Protocol.....44

4.2.4 Neuman-Stubblebine Protocol.....45

4.2.5 Needham-Schroeder with Server.....47

4.2.6 Kerberos Protocol.....48

4.2.7 Analyzing Otway-Rees.....50

4.3 Non-Regular Protocol Test Cases.....51

4.3.1 Repeating Message.....51

4.3.2 Sub Term Relationship of Encrypted Test Component.....52

4.3.3 Unreadable Encryption on Messages.....53

4.4 Summary.....53

V. Conclusions and Recommendations.....55

5.1 Chapter Overview.....55

5.1 Conclusions of Research.....55

5.3 Significance of Research.....56

5.4 Recommendations for Future Research.....56

Appendix A – Security Protocol Analyzer Code.....59

Bibliography.....89

List of Figures

Page

Figure 2.1 - Needham-Schroeder Protocol 9

Figure 2.2 - Outgoing Test 12

Figure 2.3 – Annotated Needham-Schroeder 14

Figure 2.4 - Needham-Schroeder Penetrated 17

Figure 2.5 - Kerberos Communication Initialization 20

Figure 3.1 – Class Diagram of SPA 28

Figure 3.2 – Needham-Schroeder Protocol Input File 29

Figure 3.3 – UML of Message Breakout 29

Figure 3.4 – Output of running Needham-Schroeder in SPA 37

Figure 3.5 – Output of NS with duplicate transmission of nonce 38

Figure 4.1 -Wide-Mouth Frog Protocol 42

Figure 4.2 – Yahalom Protocol 44

Figure 4.3 – Woo-Lam Protocol 45

Figure 4.4 – Neuman-Stubblebine with Tags on Keys/Tickets 46

Figure 4.5 – Neuman-Stubblebine w/o Tags on Keys/Tickets 47

Figure 4.6 – Needham-Schroeder Protocol (w/ Server) 48

Figure 4.7 – Kerberos Protocol 50

Figure 4.8 – Output of Otway-Rees protocol run 51

Figure 4.9 - Repeating Message 52

Figure 4.10 - Sub-Term Re-Encryption 53

Figure 4.11 – Needham-Schroeder with improper encryption 53

List of Tables

Page

Table 1 – Brief Description of SPA classes..... 27

Abstract

When Roger Needham and Michael Schroeder first introduced a seemingly secure protocol [24], it took over 18 years to discover that even with the most secure encryption, the conversations using this protocol were still subject to penetration. To date, there is still no one protocol that is accepted for universal use. Because of this, analysis of the protocol outside the encryption is becoming more important. Recent work by Joshua Guttman and others [9] have identified several properties that good protocols often exhibit. Termed "Authentication Tests", these properties have been very useful in examining protocols. The purpose of this research is to automate these tests and thus help expedite the analysis of both existing and future protocols.

The success of this research is shown through rapid analysis of numerous protocols for the existence of authentication tests. The result of this is that an analyst is now able to ascertain in near real-time whether or not a proposed protocol is of a sound design or whether an existing protocol may contain previously unknown weaknesses. The other achievement of this research is the generality of the input process involved. Although there exist other protocol analyzers, their use is limited primarily due to their complexity of use. With the tool generated here, an analyst needs only to enter their protocol into a standard text file; and almost immediately, the analyzer determines the existence of the authentication tests.

AUTOMATING SECURITY PROTOCOL ANALYSIS

I. Introduction

1.1 Background

When Roger Needham and Michael Schroeder first introduced a seemingly secure protocol [24], it took over 18 years to discover that even with the most secure encryption, the conversations using this protocol were still subject to penetration [9]. To date, there is still no one protocol that is accepted for universal use. Because of this, analysis of the protocol outside the encryption is becoming more important. Recent work by Joshua Guttman and others [5] has identified several properties that good protocols often exhibit. Termed "Authentication Tests", these properties have been very useful in examining protocols. The purpose of this research is to automate these tests and thus help expedite the analysis of both existing and future protocols.

1.2 Problem Statement

Numerous security protocols have been proposed [29]. They utilize both asymmetric and symmetric cryptography and employ characteristics such as trusted and non-trusted third parties. Chapter 2 covers these concepts in great detail. The problem is that analysis of these protocols is normally done either through tedious pen and paper proofs or by realizing weaknesses after the fact. This research is intended to reduce the burdensome task of evaluating protocols from a theoretical pen-and-paper method to a more automated method that incorporates techniques understood to prove certain

correctness properties of a protocol. Although there are other methods to evaluate protocols, this research focuses on the methods developed by Guttman et al [5].

1.3 Summary of Results

The Security Protocol Analyzer (SPA) successfully shows that automated tools can be highly valuable in the performance of protocol analysis. In particular, the SPA is able to determine when and where outgoing, incoming and unsolicited tests occur within a protocol run. Using string comparisons vice type comparisons requires specific values be given and does limit the application to analysis based on completed static runs. However, putting together numerous protocols in generic text files proves much easier than individual protocol development as noted in other protocol analyzers [11, 27]. It also allows for much quicker analysis of the protocol because it does not have to dynamically create a search tree; instead it only examines the post-run state of the protocol as entered in the input text file. The SPA takes any protocol as input in a standard text file and generates accurate output that shows occurrences of authentication tests. Detecting authentication tests is done in very short time.

1.4 Thesis Overview

The remainder of this thesis consists of four chapters.

- Chapter 2 – This chapter is intended to give the reader an understanding of the theoretical aspects of what this thesis intends to accomplish. This chapter focuses primarily on background literature and other work done in the field of protocol

analysis. It introduces concepts such as authentication tests, test components and different types of encryption methods.

- Chapter 3 – This chapter lays out how the protocol analysis is accomplished. It describes the inner workings of an automated analysis tool developed specifically for this research and gives the reader an understanding of how the results of the protocol analysis tool are to be interpreted.
- Chapter 4 – In this chapter, numerous protocols are executed using the analysis tool. The output is described and the importance of certain functionality, in relation to the tool, is laid out for the reader.
- Chapter 5 –This is the summary chapter. Here, the determination about the effectiveness of the tool is given. It also lays out the groundwork as to where future work in the field of protocol analysis should go.

II. Literature Review

2.1 Chapter Overview

This chapter's purpose is to give the reader a basic understanding of strand spaces and authentication tests. First, an introduction into the basic forms of encryption, symmetric and asymmetric, is given. This is necessary to ensure the reader understands basic cryptographic principles that are discussed throughout this chapter. Next, a brief introduction of what strand spaces are and how they are a means of representing protocols within the context of graph theory is given. Particularly, this chapter shows how strand spaces are used to model current protocols, such as Needham-Schroeder-Lowe [9] and Kerberos [13]. It also introduces the various authentication tests, which are derived from the theory of strand spaces. Authentication tests are a means of ensuring a protocol is designed well enough to withstand common capabilities of penetrators, such as those represented in the Dolev-Yao threat model [26].

Next, this chapter gives a brief introduction into automated modeling tools that represent potential candidates for automating the tests. The purpose of automating strand space analysis is to show whether or not potential weaknesses exist within a given protocol. If successful, this approach can provide a method for alleviating the tedium and inaccuracy associated with pen-and-paper proofs. Finally, related work in the field of strand space automation is discussed and their general results summarized.

2.2 Symmetric/Asymmetric Cryptography

A brief review of symmetric/asymmetric cryptography is warranted since there is frequent mention of 'keys' belonging to different parties throughout the remainder of this thesis.

Symmetric Cryptography

With the advent of the Caesar cipher over 3000 years ago, symmetric cryptography was established as the first form of encryption. It is the use of a single key to perform both encryption and decryption of messages. The concept works as follows:

If Bob wants to send Alice a message, they have an agreed-upon key, which they will use for encryption/decryption purposes. This key is most likely a mathematically generated prime number, which when applied to an algorithm will generate cipher text (the encrypted message). Alice or Bob can then take the cipher text, along with the same key, plug it into the same algorithm and generate the original plain text (the unencrypted message). Symmetric cryptography has the advantage of requiring only one key for both encryption and decryption, but if compromised, all messages encrypted with that one key are now in danger of being read by unintended parties!

Asymmetric Cryptography

Asymmetric encryption utilizes two keys, a private and public key to encrypt/decrypt a message. The public key is the receiver's key that is freely made available to all potential senders. The private key is the key owned by the receiver that is never shown to anyone else. In this case, Alice has a private key, which only Alice knows, and Alice has a public key, which is available to anyone. Similarly, users will entrust their public key

to Alice and keep their private key to themselves. Because Alice has another user's public key, Alice can encrypt a message with that public key. Once the message is sent, only the owner of the corresponding private key can decrypt it. Asymmetric cryptography has the advantage of guaranteeing that only the intended recipient of a message can read that message, and does not require sharing private keys with anyone. (This assumes one's private key is not compromised.) However, asymmetric keys are normally mathematically larger than symmetric keys so it is quite common for asymmetric cryptography simply to be used to encrypt symmetric keys that are distributed to intended recipients. Another more recent use for asymmetric keys is digital signing. By encrypting only a portion of the message (e.g. hash of the message) with my private key, then enclosing the total message in a symmetric key, I guarantee non-repudiation of its origin. This means that whoever opens the message is assured of its origin if the sender's private key is not compromised because at this point, only the sender's public key opens the inner encrypted portion.

2.3 Strand Spaces and Security Protocols

A strand space is a graph-theoretic representation of a security protocol. A security protocol is the handshaking that occurs between different parties, within the context of computer networks. The intent of the handshaking is to ensure authenticity of each party to the others, that authorized persons only view a message's content, and possibly to generate and/or distribute session keys. Session keys are used temporarily for encryption during time-sensitive conversations. Their advantage is that they expire; and if they are not compromised, intercepting a message from one session is not mathematically

equivalent to a message from another session regardless of the contents in the message. Their disadvantage is that they need to be generated for each new session. So, if there is a weakness in the protocol, a penetrator might be able to ascertain how session keys are being created. This very flaw is exploited in Needham-Schroeder [9].

Authentication tests stem from an understanding of strand spaces. Therefore, the first topic covered is strand spaces. Once it is shown how graphs are used to represent protocols, the next logical step is to show how messages are formed and represented along these graphs. This is where the concept of a test component is introduced. Then it is explained how outgoing, incoming and unsolicited tests are derived from test components and thus result in the formulation of the authentication tests.

2.3.1 Strand Spaces – Brief Overview

To start, let's assume for now that a protocol only consists of two parties communicating with each other. Communication consists of a series of discrete events. For example, Party A sends some sort of message to Party B. Party B receives that message and maybe sends another back, etc. This process continues for whatever length of time that given protocol requires. A *strand* is the sequence of events that occur involving only one of the parties (A's strand for this protocol: A sends a message, A receives a message, A sends another message etc). A *regular strand* is identified as a legitimate party's strand. A *strand space*, shown as Σ, is the collection of all strands, or sequences of events, that can occur between communicating parties. It is these 'strand spaces' which form the basis of authentication tests [2]. The authentication tests are a

means of verifying whether a given protocol can successfully ensure proper (intended) and secure communication between the parties.

Messages that can be exchanged between communicating parties in a protocol are called *terms*. Terms are elements of the set of messages, called *A*, which can be sent between communicating parties. The set *A* is freely generated from two disjoint sets: *T,* which represents text (nonces, names etc) and *K,* which represents keys. The generation of *A* from these sets occurs through encryption, concatenation or both. We show transmitted terms as being preceded by a positive (+) sign and received terms as being preceded by a negative (-) sign. To further illustrate this, we'll use *t* to represent a term being sent then received by party A from our example above. In the above case, we represent A sending the term as: + *t* and A receiving the term as: - *t*. Also, a term *t* is said to be a subterm of *t*', written as $t \subset t'$, if one can arrive at *t*' by "repeatedly concatenating [t] with arbitrary terms and encrypting with arbitrary keys." [8] Encryption of a term is written as: $\{ t \}_{Ka}$. If we want to show encryption from the use of a particular party's key, we write this as: $\{ t \}_{Ka}$ meaning the term is encrypted with A's public key. A's private key is denoted K_a^{-1}. To represent symmetric cryptography, encryption with a key shared by A and B is shown as $\{ t \}_{Kab.}$

Strand spaces are based on graph theory (Figure 2.1). A graph consists of edges and vertices. The vertices represent communication events, also called *nodes*. If *s* is a strand, then we represent the ith node along that strand as $n = <s, i>$. There are two kinds of edges: successive events (nodes) within a strand (shown with the double arrow: $\Rightarrow$) and communication between nodes on two separate strands (shown with the single arrow:

$\rightarrow$). The graph will be acyclic since events cannot go back and precede events that have already occurred. The relation between nodes on the same strand is represented as n $\Rightarrow^{+}$ n' where n = <*s*, *i*>, n' = <*s*, *j*> and *j* > *i*. The relationship between inter-communicating nodes, meaning nodes from separate strands, is denoted as n $\rightarrow$ n' where term(n) = +*t* and term(n') = -*t*. Finally, a *bundle* is defined as a section of the strand space that is large enough to represent a full protocol exchange. Figure 2.1 demonstrates an example of a bundle, although in this case, the bundle consists of essentially the protocol itself.

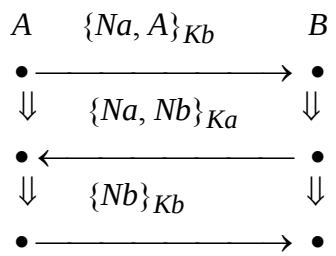

Figure 2.1 - Needham-Schroeder Protocol

This brief introduction into the notation used in strand space analysis suffices to demonstrate how a protocol is represented. Later in this chapter, this notation is used to show the Kerberos protocol (Figure 3) in detail. Next, incoming and outgoing tests are described. These tests are the foundation of the authentication tests.

2.3.2 Understanding the Penetrator

First, to help understand why the authentication tests are important, it is now necessary to explain what the tests help protocol designers guard against. The penetrator is understood to be the person(s) who is trying to perform any unwanted action during an exchange. Dolev-Yao [26] have formalized what are understood to be widely accepted

capabilities of the penetrator. This formalization is termed the Dolev-Yao threat model. Because the focus here is not on the mathematical soundness of encryption, it is understood that the penetrator is actually a legitimate party on the network simply out to do no good, so we term this person: Malice. What can Malice do?

- Malice can obtain any message passing through the network
- Malice is a legitimate user and can initiate conversations, and is expected at one time, to be a recipient of an initiated conversation
- Malice can impersonate any principal and thus send messages on their behalf to any other principle on the network

Dolev-Yao also explicitly state what Malice can not do:

- Malice cannot guess a random number (i.e. the mathematics of the encryption is assumed to be ideal)
- Malice cannot decrypt properly encrypted messages without possessing the proper key; Malice cannot generate encrypted text on behalf of a user without his or her proper key
- Malice cannot ascertain the correct corresponding private key of any other user's public key

2.3.3 Authentication Tests

Now that we understand what we are guarding against, we can move onto the authentication tests themselves. To understand the authentication tests it is necessary to understand three simple tests: outgoing, incoming and unsolicited [4]. It is these three tests that form the foundation of the authentication tests.

The key component that these tests work with is called a test component. The formal definition of a test component is shown below [4]:

Definition: $t = \{h\}_k$ is a test component for a in n if:

1. $a \subset t$ and t is a component of n;
2. The tem t is not a proper subterm of a component of any regular node $n' \in \sum$.

What this states in laymen's terms is that a principal generates some *a,* and it is *a*'s existence in a component that differentiates between a routine component and a test component. The transmission or reception of this test component is how we ascertain whether an incoming, outgoing or unsolicited test occurred.

An "outgoing test for *a* in *t*" is when a test component *t* that contains a uniquely originating value *a* is sent out and *a* is received back in cryptographically altered form called *t'* (Figure 2.2); (cryptographically altered form means that the initial message is decrypted by someone possessing the proper key and subsequently altered) the conclusion is that an authorized recipient received the message, decrypted it, extracted the value *a* and transmitted it back. This conclusion relies on the assumption that the decryption key, K^{-1}, is safe, or not compromised by an attacker, and therefore only a regular (authorized) user could perform the decryption. The uniquely originating value, in this case *a,* which is a very large randomly generated number, has very little chance of being guessed by another party. The uniquely originating term is indicated by the * and

we represent the outgoing test as: $\{\ldots a \ldots\}_K \sim \ldots a \ldots$ where a is the uniquely originating value.

$$\begin{array}{ll} \text{A} & *a \subset \{\text{h}\}_K = \text{t} \\ \bullet & \xrightarrow{\qquad\qquad} \\ \Downarrow & a \subset \text{t}' \\ \bullet & \xleftarrow{\qquad\qquad} \end{array}$$

Figure 2.2 - Outgoing Test

Coincidentally, the creation of this unique number is very common. A regular use of this unique value is as a nonce or numbers once; because of their size and randomness, they are commonly used as session keys. The unaltered portion of the message is this uniquely created value because it is possible that the intended recipient concatenates other values to the original message. With regard to the graph representation, the part of a strand that receives *a* and sends it back altered is referred to as a *transforming* edge. The part of a strand that sends *a* out and receives it back altered is referred to as a *transformed* edge. A transformed edge containing a uniquely originating term in the sending node is called a *test*.

The incoming test works in a similar fashion. Given some *a* transmitted in either plain or encrypted form, if it is received back unaltered but within a test component properly encrypted by an uncompromised symmetric key, we conclude that a regular recipient performed the encryption. We write this as: $\ldots a \ldots \sim \{\ldots a \ldots\}_{Kab}$. The unsolicited test is inherently weaker in nature. It states that whenever a test component $\{\ \text{t}\ \}_K$ is received, assuming that symmetric key *K* is safe, then the term could only have originated on a *regular* strand. Since the graphical representation of the protocol is acyclic, this

originating node is located somewhere before the reception node. We show this as: ~ $\{\ldots a \ldots\}_{Kba}$. These three tests provide the groundwork for understanding how the authentication tests work. Since the purpose of this research is to generate a tool that automates security protocol verification, the following are the formal definitions for authentication tests and are drawn directly from [2]:

Authentication Test 1: Let C be a bundle with $n' \in C$, and let $n \Rightarrow+ n'$ be an outgoing test for a in t.

1. There exist regular nodes m, $m' \in C$ such that t is a component of m and $m \Rightarrow+ m'$ is a transforming edge for a.

2. Suppose in addition that a occurs only in component $t1 = \{h1\}K1$ of m', that $t1$ is not a proper subterm of any regular component, and that $K1^{-1} \notin P$. Then there is a regular node with $t1$ as a component.

Authentication Test 2: Let C be a bundle with $n' \in C$, and let $n \Rightarrow+ n'$ be an incoming test for a in t'. Then there exist regular nodes m, $m' \in C$ such that t' is a component of m' and $m \Rightarrow+ m'$ is a transforming edge for a.

Authentication Test 3: Let C be a bundle with $n \in C$, and let n be an unsolicited test for $t = \{h\}K$. Then there exists a positive regular node $m \in C$ such that t is a component of m.

2.3.4 Tests Applied to Needham-Schroeder

This section shows how these tests are visible in an actual Protocol. Although the Needham-Schroeder is described later on in great detail, it serves as a good tool to further understand how the tests work.

$$
\begin{array}{ccccc}
 & A & \{Na, A\}_{Kb} = \boldsymbol{t} & B & \\
 & \bullet & \longrightarrow & \bullet & \\
\text{“}\textbf{\textit{Transformed Edge”}} \rightarrow & \Downarrow & \{Na, Nb\}_{Ka} = \boldsymbol{t'} * & \Downarrow & \leftarrow \textbf{\textit{”Transforming Edge”}} \\
 & & & & \\
 & & \textbf{\textit{*= t' w/ respect to A,= t w/ respect to B}} & & \\
 & & & & \\
 & \bullet & \longleftarrow & \bullet & \\
\textbf{\textit{”Transforming Edge”}} \rightarrow & \Downarrow & \{Nb\}_{Kb} = \boldsymbol{t'} & \Downarrow & \leftarrow \textbf{\textit{”Transformed Edge”}} \\
 & \bullet & \longrightarrow & \bullet &
\end{array}
$$

Figure 2.3 – Annotated Needham-Schroeder

Working from Figure 2.3, in the first line, A is sending his test component out. The message {A Na}Kb is valid as a test component for reasons described in Section 2.3.3. (The *a* in this example is represented by *Na*.) The component is received by B, who then transforms the term through decryption, alteration and re-encryption (only the end result is shown). With respect to the SSM, this constitutes the transforming edge. A's receipt of this new component, with respect to the SSM, is the transformed edge. The receipt of this new component in this protocol represents two actions: The first is the completion of A's outgoing test and the second is B's transmission of his own test component. Because A has received his test component back altered, only through proper decryption, he can infers a proper principal performed the transformation thus completing his outgoing test. A then takes over the role as 'transforming edge' with respect to B's test component: {Na

Nb}Ka, with the *a,* for B's test component, represented as *Nb*. A decrypts, alters and re-encrypts B's test component, then retransmits the new component back to B. Once B receives his new component back, he goes through the same evaluation proofs as A and thus concludes a proper principal performed the transformation and concludes an outgoing test has occurred.

2.4 Security Protocol Examples

The strand space methodology enables the modeling of security protocols as graphs. This section describes two security protocols represented in the context of strand spaces. The two protocols are: Needham-Schroeder [2] and Kerberos [13]. However, in an effort to illustrate the use of strand spaces with a particular protocol, more emphasis is placed on the Needham-Schroeder protocol as this protocol is routinely studied and analyzed in the context of strand spaces [2, 4 and 5].

2.4.1 Types of Protocols

Clark and Jacob identify basic categories that protocols fit into: Symmetric or Asymmetric cryptography, employing either trusted third parties or simply two communicating principals [29]. For the purpose of this research only the main three protocol types are reviewed. They are: Symmetric Key with Trusted Third Party, Symmetric Key without Trusted Third Party and Public Key. Symmetric key with trusted third party is demonstrated in the Kerberos protocol. In this protocol, the session keys are generated by a server and then distributed to the requesting parties. There exists an

understanding that each party has its own secure symmetric key for use when communicating with the server.

The symmetric key without trusted third party is best demonstrated in challenge-response protocols. The way Needham-Schroeder is shown in Figure 2.1, it could be construed as a challenge response protocol had the keys been symmetric vice asymmetric. The reason is, had the keys been symmetric, it would imply that the communicating parties each had prior knowledge of the key they intend to use with no server generating it for them. However, Needham-Schroeder does use asymmetric cryptographic techniques with a trusted third party.

2.4.2 Needham-Schroeder Protocol

The Needham-Schroeder [24] public key protocol is represented in Figure 2.1. Later in Chapter 4, this protocol is shown with the server. A description of the protocol represented in Figure 2.1 is as follows: The parameters *A* and *B* represent communicating principals. The parameter *N* represents a nonce. The letter following *K* is indicating which node's public key is used. In the Needham Schroeder protocol, *A* sends *B* a nonce encrypted with *B*'s public key. Along with this, *A* sends *B* his signature; in this case a signature is simply some agreed upon identifier which each party can use to know who they are speaking to. Node *B* decrypts the message and replies by sending *A* the original nonce along with a new nonce generated by *B*, all encrypted with *A's* public key. Node *A* decrypts the message and sends *B* his nonce back, encrypted with *B's* public key. Once these events happen, it is now understood that *A* and *B* are communicating. However, it is not implied that this is a secure sequence of events. In fact, it is shown [9,

2 and 5] that strands termed *penetrator strands* are capable of infiltrating a session. The *penetrator strand* represents an unauthorized party who has infiltrated a session. The effect of infiltration varies significantly based upon the importance or sensitivity of the session.

2.4.3 Penetration of Needham-Schroeder Defined

The penetration of the Needham-Schroeder occurs as follows [Figure 2.4]: Assume that *A* wants to talk to another party. In this case, we'll call that party *P*. *A* then initiates a conversation with *P*, who then encrypts/forwards *A*'s information over to party *B*. Party *B*, thinking *A* initiated a conversation with him, will then answer the challenge with a reply, and also issue his own challenge. *A* receives this message and assumes it came from *P* and then replies to *P* with the correct response. *P* then encrypts/forwards this new information over to *B* who sees it as the correct information.

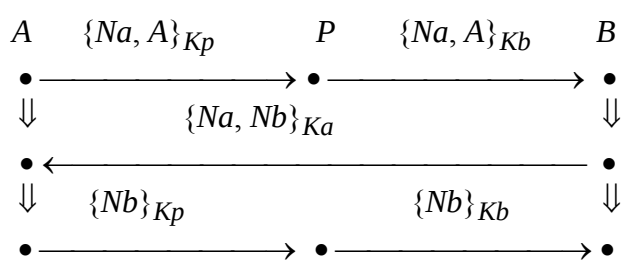

Figure 2.4 - Needham-Schroeder Penetrated

Now, both parties are convinced they are talking to their intended audience. However, *P* now has the nonces from each party and is able to eavesdrop on a conversation between *A* and *B* or simply converse with *B* while impersonating *A*. In [9], Lowe proposes that *B*'s reply contain his identifier: ({ *Na, Nb, B*}*Ka*). This will allow *A* to see that

somehow his information is going to an unintended party and deduce that *P* is probably malicious.

2.4.4 Kerberos

Kerberos is an authentication service for open network systems first proposed by Miller and Neuman [14]. It involves three principals: user, client and server. The basic process works as follows. A user desiring a service or program contacts the client. The client can be anything from a program to a person. The client then contacts the server on behalf of the user. Kerberos, using private key encryption, derives a private key from the users' password. This key, along with all other users' keys, is stored in a Kerberos-managed database. Any network service requiring authentication is registered with Kerberos. Also important to note, Kerberos maintains a list of registered clients, which corresponds with a particular key. What is evident here is that Kerberos acts as the middleman between registered clients. Another role Kerberos plays is the generation of session keys. Unlike the private keys, session keys are only temporary keys used within a limited timeframe, as stated earlier; nonces are often used for session keys.

It is now demonstrated how Kerberos handles a request for communication between two parties. *A* first requests a session key from the authentication server that it can then use to communicate with *B*. The session key is a uniquely originating key, which is ideally used for only one session. First, *A* sends the authentication server, *AS*, its identity and the identity of whom it wants to communicate with. *AS* then generates a symmetric session key, it encrypts the session key and *B*'s identity with *A*'s key. It then encrypts the session key and *A*'s identity with B's key (Figure 2.5). It then sends both encrypted

messages to *A*. At this point, *A* decrypts the session key encrypted with *A*'s key, generates a time stamp, encrypts the time stamp with the newly acquired session key and sends both packages off to *B*. Now, *A* and *B* are able to communicate in a secure fashion.

With respect to the authentication tests, Kerberos uses the incoming and unsolicited tests. This determination is made by the fact that *A* sends out plaintext and receives it back properly encrypted. This is an example of the incoming test. With respect to incoming tests, sending something out plain and receiving it back encrypted does not imply symmetric cryptography; if that protocol uses asymmetric cryptography then any party can encrypt a plain message with a public key and one cannot conclude an incoming test occurred because you cannot be sure who did the encrypting. However in Kerberos, *A* shares a symmetric key with *AS* therefore we conclude the *AS* did the encrypting. *B* receives an un-requested message properly encrypted by *A*; this is an example of an unsolicited test. This protocol does not contain any outgoing tests, but it is not a requirement that all tests be represented in a protocol. The presence of these tests also does not guarantee that Kerberos is a 'good' protocol. The original Needham-Schroeder contained outgoing tests with respect to both parties but still demonstrated a serious flaw due to the contents of the messages themselves.

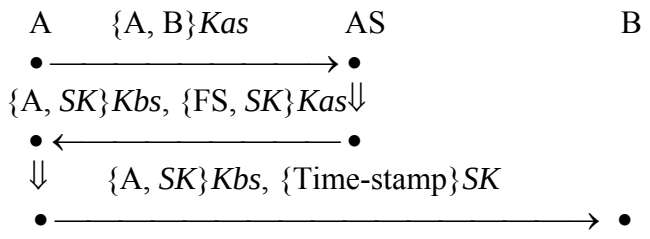

Figure 2.5 - Kerberos Communication Initialization

2.5 Related Work in the Automation of Security Protocol Verification

Automating security protocol verification allows for quicker determinations of the weaknesses and strengths of a particular protocol. Numerous researchers [11] have proposed applying formal proof techniques towards the analysis of security protocols. The next sections briefly cover key work that has been successful in the automation of security protocol verification.

The techniques employed in protocol analysis, covered in this review, typically fall into two categories. They are model checking and theorem proving. In the model checking approach, one searches for desired states by modeling the protocol and executing it in every possible way. In the theorem proving approach, one creates a search tree and checks for the existence of the theories in that tree at some state. The Multi-Set Rewriting and Failure Divergence Refinement [22] checking typically fall into the model checking camp. The Athena approach [11] falls into the theorem proving camp. The approach in this research falls into the theorem approach because it automatically identifies where the authentication test theorems can be applied.

2.5.1 Failure Divergences Refinement Checker

Lowe [9] successfully analyses the Needham-Schroeder protocol using the Failure Divergences Refinement Checker (FDR), which is a model checker for Communicating Sequential Processes (CSP) [25]. CSP is a language that allows easy representation of pattern interaction. Using CSP, Lowe tests whether a protocol achieves authentication. In the case of an attacker being discovered [9], Lowe is also able to show that the fixed protocol is secure.

2.5.2 Process Calculus

In [1], Blanchet uses an extension of pi-calculus to represent protocols. Pi-calculus is "a 'process algebra' in which channel names can act both as transmission medium and as transmitted data" [23]. His results show promise in that automated protocol verification can be done in less than one second. The user needs only to correctly code whatever protocol they intend to evaluate. The tool, OCaml 3.04 [1], translates the protocol into Horn clauses and then executes it against rules based on whatever that particular protocol defines as a rule. This is obviously much faster than traditional methods of hand proving. Blanchet also permits an unbounded number of sessions within the test, whereas previous work has limited the number of session due to infinite state systems. On rare occasions the algorithm will not terminate and it can even fail on correct protocols. However, it is Blanchet's conjecture that it will terminate for a large class of protocols.

2.5.3 Athena

Athena is an automated checking algorithm that analyzes security protocols [12]. Athena implements a specialized logic for expressing security properties such as authentication, secrecy and properties related to electronic commerce [12]. Athena works by terminating and providing a proof on well-formed formulas or generating a counter example on well-formed formulas that evaluate to false. Although there are other formal techniques for analyzing security protocols [11], Athena differs in the sense that it can directly evaluate the strand space model without succumbing to the state space explosion [11, 2.5.1]. State space explosion occurs whenever there is an unbounded number of initiators/responders and sessions thus creating an unreachable theorem [18]. Athena uses 'unreachability theorems' to 'prune' the state space, thus reducing the number of states and increasing the probability of terminating [12]. Another method Song uses to reduce the state space is that of the Strand Space Model (SSM) [28]. By using the causal relationships developed by Guttman in the SSM, Song is able to further prune the state space. Song shows that although limited due to its inability to allow certain terms to be encrypted, Guttman and Thayer's tests are still effective in reducing the state space.

2.5.4 Security Modeling in Maude

Object modeling software is another effective way to model protocol transactions. One example of automated modeling software is Maude [19]. Maude was the intended language for this research, but Java proved very effective. However, for future research it is highly recommended that using Maude to automate authentication tests be performed, as this is now explained. Developed primarily at the University of Illinois

and Stanford Research Institute, Maude's purpose is stated as follows: "…supporting formal executable specification, declarative programming, and a wide range of formal methods as means to achieve high-quality systems in areas much as: software engineering, networks, distributed computing, bioinformatics, and formal tool development." [19].

Specifically, Maude is a "high-performance reflective language, which supports both equational and rewriting logic specifications" [19]. Being a reflective language means that Maude can program or manipulate itself [20]. Rewriting logic allows for concurrent state computations. Although a protocol is pre-defined at run time, concurrent state computations allows for rules or equational rewrites to occur simultaneously.

In [27], Denker and Meseguer show how protocols can be effectively modeled using object-oriented specification in Maude. The purpose of his work is to show that Maude's ability to perform rewritable logic and concurrent execution is helpful in uncovering security flaws in protocols. The paper uses the Needham-Schroeder public key protocol as the case study. Meseguer implements a bounded depth first search on multiple instances of protocol runs. The depth first search shows efficient searching of possible attacks on the multiple run is an effective means of ascertaining a weakness on a given protocol. In this case, the weakness discovered by Lowe [9] is found using a depth first search. In this example we see Maude being used as a model checker. However, Maude can also be used for theorem checking. Because Maude is powerful enough to handle either method of use, it allows for multiple means in which modeling the authentication tests can be done.

2.5.5 Multi-Set Rewriting

In [22], Cervesato et al. use multi-set rewriting to provide a means to specify 'finite length' protocols. In his work, Cervesato uses the multi-set rewriting to extend the strand space formalism. Through this he is able to model penetrator capabilities, as defined by Dolev-Yao, and then relate the intruder theory to penetrator strands as defined within the context of strand space modeling. This particular work serves as a means to further understand the Dolev-Yao threat model.

2.6 Summary

This chapter outlines what a strand space is and how it is represented using graph theory notation. It then shows how this notation is used to represent a security protocol. The Needham-Schroeder and Kerberos protocols were used to illustrate the use of strand spaces. Next, three authentication tests were defined from [5]. There are three tests—outgoing, incoming and unsolicited—and each has a theorem describing the guarantees it provides. Finally, previous work in automating aspects of security protocol analysis has been discussed.

III. Methodology

3.1 Chapter Overview

The purpose of this chapter is to introduce the methods that are employed in automating the authentication tests developed by Guttman et al [2, 5]. This chapter describes the Security Protocol Analyzer (SPA) tool developed for the purpose of this research. The chapter also shows SPA's analysis of a representative set of known protocols and basic examples in order to establish a baseline of reliability. In Chapter 4, a much more diverse set of protocols is analyzed.

This chapter is laid out in the following manner:

- Problem Review
- Overview of software used
- Java based protocol analysis
- Demonstration of tests in the following environments:
    - Numerous one and two pass tests
    - Needham-Schroeder
- Summary

3.2 Problem Review

The problem that this research addresses is the effective automation of security protocol analysis, and in particular, the automatic recognition of authentication tests as defined by Guttman et al. Guttman et al have developed three authentication tests that greatly simplify the tedious pen-and-paper proofs normally required to show if a security

protocol has certain correctness properties. This research takes the next step and automates aspects of their work. In the previous chapter, these tests and correctness properties have been defined and now the focus will shift to how occurrences of these tests are automatically identified using a tool developed in Java.

3.3 Software Overview

The Java-based protocol analyzer was developed using the Java development environment TogetherSoft® version 6.0. This application was executed on a Windows® 2000 operating system. The version of Java that TogetherSoft® 6.0 employs is 1.3.1.

3.4 Java Based Protocol Analysis

The protocol analyzer uses a traditional software development approach of object-oriented programming. The analyzer uses dynamically created objects to represent different component classes, such as Principals, Messages and Text. The next few sections outline the algorithm and the specific techniques used in the creation of the tool.

3.4.1 Layout of Java Program

The Java-based program, called Security Protocol Analyzer (SPA), can be divided into two portions. The first portion is the parsing portion. The second portion is as the analysis portion.

Table 1 – Brief Description of SPA classes

Class:	**Brief Description:**
ProtocolAnalyzer	Main driver of program. Contains m*ain*().
Encryption	Checks encryption of messages against receiver and sender. Ensures proper parties are viewing messages.
Protocol	This is the root node of an abstract syntax tree representing a protocol. It contains a list of Messages.
Message	An instance of message is instantiated as each message is read in from the input file. A message consists of a sender, a receiver and the term sent.
Sequence	Since messages exist as concatenations of terms, we call this a sequence. This class contains an iterator which is traversed during analysis on instances of Sequence.
Term	The primary abstract class that enables Text, Encryption, and Sequence to generically create functions for recursive use. Two key methods for analysis, GetReadableText() and GetTextObj(), are established here.
Text	This class represents individual instances of each text. If tagged with a * in the input file, a flag condition is set to show its new/fresh characteristic.
Parser	This class checks the syntax of the input file and verifies if it is legal or not. It also builds an abstract syntax tree rooted at an instance of the Protocol class.
Principal	For each occurrence of a party within a protocol transaction, an instance of principal is generated. Within it, the test conditions are checked within the addnonce() method as components get passed.

The main role of the parsing portion is to retrieve a file and import the contents into "iterators" that the analyzer steps through and analyzes. It also builds the abstract syntax tree (AST), which is traversed during the analysis portion. The parsing method could have been done numerous ways therefore not much attention is given to the specifics in this thesis but the authentication tests are strictly defined and hence that is where our focus lies. Table 1 lays out brief descriptions of key classes in the SPA. Sections 3.4.2

and 3.4.3 go into detailed specifics of the more important operations and give a more thorough description of the key classes as seen in the SPA class diagram, Figure 3.1.

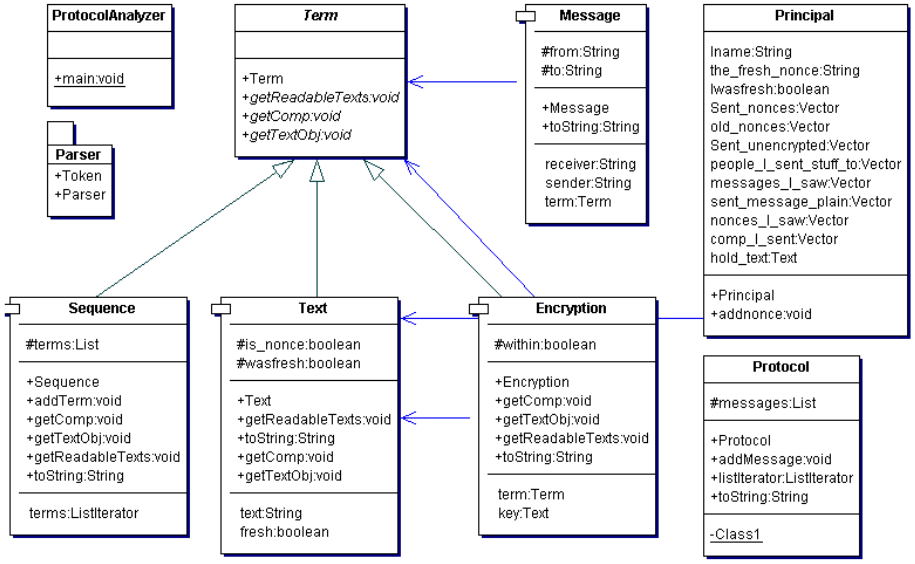

Figure 3.1 – Class Diagram of SPA

3.4.2 Description of Parsing Operation

The parsing of the SPA is performed as follows: An input file (Figure 3.2) is parsed line by line into an abstract syntax tree with a Protocol object as the root. A Protocol consists of a list of Message objects. Each Message contains three portions: *from*, *to* and the *message* itself. It is this list of messages that we walk through during the analysis portion.

A -> B : {A *Na1}Kb
B -> A : {Na1 *Nb1}Ka
A -> B: {Nb1} Kb

Figure 3.2 – Needham-Schroeder Protocol Input File

Figure 3.3 shows an example of the abstract syntax tree created during the initial parsing of a message. As seen in Figure 3.3, whenever a message is broken down, an instance of Sequence, Encryption or Text is created. Due to the recursive structure of message terms, within Encryption and Sequence lie further instances of Sequence, Encryption or Text. The breakdown process continues recursively until individual Text instances are all that remain. The parsing of the input file into AST's completes the initial run of the SPA with respect to the input file. The next operation performed is the analysis portion.

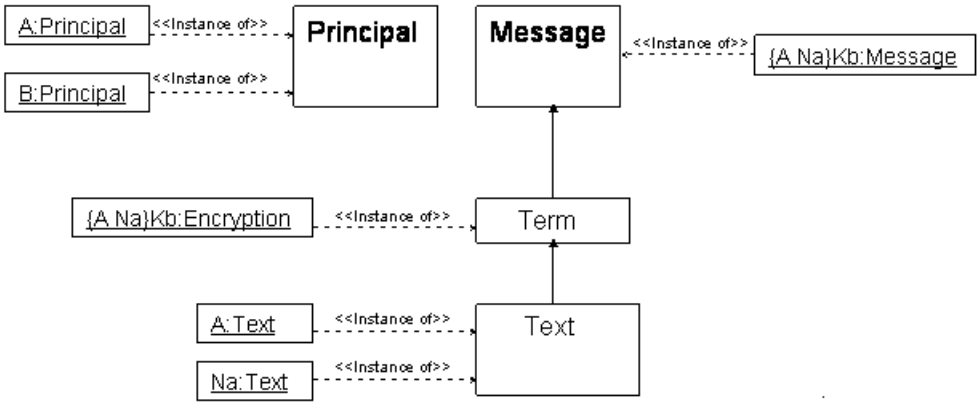

Figure 3.3 – UML of Message Breakout

3.4.3 Description of Analysis Operation

This section covers the key classes that perform the main analysis. The ProtocolAnalyzer, Principal and Term classes are what drive the analysis portion of the

program. The ProtocolAnalyzer class contains the *main()* function and hence is the driver for the program. The program makes several passes through the list of messages during the analysis portion.

The first pass through the message list checks for potentially disqualifying conditions. For instance, one of the more common conditions is occurrence of an encrypted term within another encrypted term, thus negating the inner one's use as a test component in accordance with criteria outlined by Guttman [5] and described in Chapter 2. The SPA takes this encrypted term and stores it in a vector with any other disqualified encrypted terms. If an encrypted term is not disqualified it is stored in another vector which contains potential test components. Later on, these vectors are viewed to determine if a component received by a Principal is disqualified as a test component or not.

The second pass through the message list creates instances of the Principal class and stores them in a vector. Recall that during the parsing process, each line of the input file generates a *from, to* and *message*. The *from* and *to* are used to create these instantiations of Principal. For example, using the example in Figure 3.2, in Figure 3.3 A and B are created as different instances of Principal. However, the parser first checks to make sure a previous existence of that Principal does not exist, that way only one instantiation per principal occurs. By having individual instantiations for each participant in the protocol, this allows each party to know what terms and components it has sent and received.

The third pass is where actual analysis of individual terms occurs. Using Figure 3.2 as an illustration, whenever the first line is analyzed, individual terms A and Na (parsed into an AST as described earlier), are stored individually in a sent items vector of A's

instantiation of Principal. B's instantiation of Principal will store them in a received items vector. As the SPA iterates through the list of messages, each principal, depending on its role as sender or receiver, checks its vectors of sent and received items and then applies the rules of the tests against that item. With respect to Figure 3.2, the SPA output generated is shown in Figure 3.4. This serves as a good overview regarding the analysis operation but there are two key functions that allow this to operate so effectively.

The first key function is the *GetReadableText*() function. The analyzer calls the *GetReadableText*() function initially in *main*() to initiate the third pass, but as described shortly, the SPA actually traverses the tree through recursive calls of *GetReadableText*(). This recursion happens because the *GetReadableText*() function is an abstract function stemming from abstract class Term. In order to allow more effective analysis without duplication of functionality, Term was developed as an abstract class from which Encryption, Text and Sequence are all extended. As the analyzer starts at the top of the AST, based on whatever instantiation lies at that particular node, the SPA knows what class' *GetReadableText*() function to call. From within this Term, it steps down to the next node and the SPA then calls that particular instance of *GetReadableText*(). This is especially important for encrypted terms. If the analyzer is on an encrypted node, it will determine if the receiver can view the message through string comparisons on the key and Principals identity (For example: Ka is viewable by A only). As the analyzer steps further down the tree, based on the results of whether a Principal can read the encrypted portion or not, the analyzer will know which information to pass into that Principals instantiation, thus, a received item, if not readable, will not be entered into a receiver's

vector of received terms! (The actual addition of individual text into a principals' relevant vectors is done through the *addNonce*() function, described shortly.) Once the end node of the AST is reached, we know from earlier that this will be an instance of an individual Text. At this point, when *GetReadableText*() is called, the values of the arguments at this point are now ready to be added directly into each Principal's relevant vectors.

Within the Text class' *GetReadableText*() function is called the other key function: *addNonce*(). *AddNonce*() is a method of the Principal class that is only called from within each individual instance of Text. In *addNonce*(), the following arguments get passed in: the vectors of bad test components, good test components, the name of the individual Text currently being analyzed, as determined by being at the bottom of the AST, and whether or not the sender/receiver can read the individual Text. *GetReadableText*() of that particular instance of Text will traverse the vector of Principals then whenever sender and receiver get matched, that particular Principal has its *addNonce*() function called. It is within *addNonce*(), since we are getting all the key information at this time for this particular Text, that the existence of the tests with relation to this particular Text are determined. The determination of the existence of tests follows strictly the criteria defined by Guttman [5] and covered extensively in Chapter 2.

3.5 Specific Protocols analyzed using Java

In this section, the SPA is tested with simple input to demonstrate its ability to find examples of the authentication tests under a variety of basic conditions. For example, simple one-pass and two-pass runs are performed using both asymmetric and symmetric

cryptography. The chapter then culminates with the SPA running the Needham-Schroeder protocol in both a normal setting and a setting with intentional flaws. Once these tests are completed, a reliable baseline is established for use of the SPA in identifying occurrences of outgoing, incoming and unsolicited tests in unfamiliar protocols.

3.5.1 Verifying Presence of Tests

In the first example, A executes an incoming test. A sends out plain text and receives it back properly encrypted with a symmetric key, thus completing an incoming test. The layout for all tests is similar in nature. The SPA shows encryption keys used then proceeds to identify the existence of any tests or errors.

File Contents:
A -> B : *Na1 A
B -> A : {Na1 B}Kab

<Parties> : <Message> >> A -> B : *Na1 A
Sender may be attempting to initiate an incoming test by transmitting Na1 in the clear.

<Parties> : <Message> >> B -> A : {Na1 B}Kab
Encrypt term(s) < Na1 B > with key Kab is readable by both sender/receiver.
The unencrypted/fresh nonce Na1 has been received back in new component: {Na1 B}Kab
Incoming test for A because fresh term Na1 was sent out earlier in < Na1 >

In this example, a simple two-pass run of an outgoing test is demonstrated:

File Contents:
A -> B : {*Na1 A}Kab
B -> A : Na1 B

<Parties> : <Message> >> A -> B : {*Na1 A}Kab
Encrypt term(s) < *Na1 A > with key Kab is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.
Unsolicited test for B because of nonce Na1 within test component < {Na1 A}Kab >

<Parties> : <Message> >> B -> A : Na1 B
The encrypted/fresh nonce Na1 has been received back in new component: Na1

Outgoing test for A because fresh term Na1 was sent out earlier in < {Na1 A}Kab >

In the first pass, the SPA identifies what key is used for encryption and states which parties can view that part of the message. Then, it states that the sender may be initiating an outgoing test. Also, based on the use of symmetric cryptography, as indicated by the key format, it also indicates the presence of an unsolicited test from B's perspective. B transforms the message through decryption, once again, assuming the key is safe, then retransmits it back to A. Finally, it shows A completing an outgoing test because he is receiving his fresh nonce back in altered form.

In the below example, a simple case of nested encryption is used to demonstrate another example of an outgoing test. In this case, A's fresh nonce is inside an encrypted term, which is itself encrypted. On the first pass, the SPA shows both terms' encryption keys and states how A is attempting to initiate an outgoing test. On the second pass, the SPA verifies that A has completed his outgoing test. Notice on this pass there is no unsolicited test because the communicating parties are using asymmetric cryptography, as indicated by their key format. Recall from Chapter 2, all public keys are assumed to be compromised. Also, although not shown by the SPA, the validity of this test depends on B's private key not being compromised.

File Contents:
A -> B : A {{*Na1}Ka}Kb
B -> A : {Na1}Ka

<Parties> : <Message> >> A -> B : A {{*Na1}Ka}Kb
Encrypted term(s) < {*Na1}Ka > with key Kb is readable by recipient only.
Encrypted term(s) < *Na1 > with key Ka is readable by sender only.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.

<Parties> : <Message> >> B -> A : {Na1}Ka
Encrypted term(s) < Na1 > with key Ka is readable by recipient only.

The encrypted/fresh nonce Na1 has been received back in new component: {Na1}Ka
Outgoing test for A because fresh term Na1 was sent out earlier in < {{Na1}Ka}Kb >

The next example is a more common example of what one would expect during a routine transaction using symmetric cryptography. In the example, it shows the unsolicited test that exists due to the symmetric cryptography used, as indicated by the key format used, but more importantly because of the valid test component received by B. The only thing to note here is the phrase incoming/outgoing test. This is not a new test. Recall from Chapter 2, the outgoing test is sent out encrypted and received back altered, possibly unencrypted; the incoming test is sent out in either format but received back encrypted. Therefore, this example fulfills the requirements of both tests and could be called either, thus it is termed an outgoing/incoming test!

File Contents:
A -> B : {*Na1 A}Kab
B -> A : {Na1 B}Kab

<Parties> : <Message> >> A -> B : {*Na1 A}Kab
Encrypt term(s) < *Na1 A > with key Kab is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.
Unsolicited test for B because of nonce Na1 within test component < {Na1 A}Kab >

<Parties> : <Message> >> B -> A : {Na1 B}Kab
Encrypt term(s) < Na1 B > with key Kab is readable by both sender/receiver.
The encrypted/fresh nonce Na1 has been received back in new component: {Na1 B}Kab
Outgoing/Incoming test for A because fresh term Na1 was sent out earlier in < {Na1 A}Kab >

In this next example, it appears as though there should be a simple incoming test. However, this is not the case. A sends out an unencrypted fresh term and receives it back encrypted. The SPA indicates that principal A may be attempting to initiate an incoming test; however, because everyone knows public keys, A cannot be sure who encrypted his plain text. The SPA's adherence to the notion of compromised public keys, as discussed in Chapter 2, disallows a completed incoming test.

File Contents:
A -> B : *Na1 A
B -> A : {Na1 B}Ka

<Parties> : <Message> >> A -> B : *Na1 A
Sender may be attempting to initiate an incoming test by transmitting Na1 in the clear.

<Parties> : <Message> >> B -> A : {Na1 B}Ka
Encrypted term(s) < Na1 B > with key Ka is readable by recipient only.

The final example is very similar to the earlier one in which a term is sent out encrypted and received back altered. This represents an outgoing test for A because test component {*Na1 A}Kb is transmitted to B. B receives the test component, based on the input file composition, transforms the test component into {Na1 B}Ka and transmits it back to A. Since encryption is correct and private keys are assumed safe, we see a valid instance of an outgoing test.

File Contents:
A -> B : A {*Na1 A}Kb
B -> A : B {Na1 B}Ka

<Parties> : <Message> >> A -> B : A {*Na1 A}Kb
Encrypted term(s) < *Na1 A > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.

<Parties> : <Message> >> B -> A : B {Na1 B}Ka
Encrypted term(s) < Na1 B > with key Ka is readable by recipient only.
The encrypted/fresh nonce Na1 has been received back in new component: {Na1 B}Ka
Outgoing test for A because fresh term Na1 was sent out earlier in < {Na1 A}Kb >

3.5.2 Analyzing Needham-Schroeder

The above examples showed different examples of the outgoing, incoming and unsolicited tests. In this section we expand upon this by running an actual protocol through the SPA. The result of normally operating the Needham-Schroeder protocol in the SPA confirms the previously known existence [5] of outgoing tests in the protocol. Figure 3.4 shows the entire output for the original Needham-Schroeder protocol.

```
A -> B : {*Na1 A}Kb
B -> A : {Na1 *Nb1}Ka
A -> B : {Nb1}Kb

<Parties> : <Message> >> A -> B : {*Na1 A}Kb
Encrypted term(s) < *Na1 A > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.

<Parties> : <Message> >> B -> A : {Na1 *Nb1}Ka
Encrypted term(s) < Na1 *Nb1 > with key Ka is readable by recipient only.
The encrypted/fresh nonce Na1 has been received back in new component: {Na1 Nb1}Ka
Outgoing test for A because fresh term Na1 was sent out earlier in < {Na1 A}Kb >
Sender may be attempting to initiate an outgoing test by transmitting Nb1 in encrypted form.

<Parties> : <Message> >> A -> B : {Nb1}Kb
Encrypted term(s) < Nb1 > with key Kb is readable by recipient only.
The encrypted/fresh nonce Nb1 has been received back in new component: {Nb1}Kb
Outgoing test for B because fresh term Nb1 was sent out earlier in < {Na1 Nb1}Ka >
```

Figure 3.4 – Output of running Needham-Schroeder in SPA

Although there are no surprises here, it is important that an actual protocol was run. From this, we are now able to move forward and test protocols where outcomes may not necessarily be expected. The final test is to run the Needham-Schroeder protocol in slightly altered form, then using already known results [5], verify that the analyzer generated the correct results.

In this next test we present an altered form of Needham-Schroeder. In Figure 3.5, we see several different erroneous/mischievous activities being performed. The first is principal C trying to resend a previous message. The SPA generates no notice on this because it cannot assume this was intentional. It may simply be acting as a relay. From the legitimate test, it recognizes that A has received his fresh nonce back and therefore won't reregister this as another outgoing test. The next activity is that of B trying to initiate a conversation using A's old nonce; once again no test is registered. These duplicate transactions do occur several more times but to no avail.

A -> B : {A *Na}Kb
B -> A : {Na *Nb}Ka
C -> A : {Na Nb}Ka
B -> A : {B Na}Ka
A -> B : {Na}Kb
A -> B : {Nb}Kb
A -> C : {Nb}Kc

<Parties> : <Message> >> A -> B : {A *Na}Kb
Encrypted term(s) < A *Na > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Na in encrypted form.

<Parties> : <Message> >> B -> A : {Na *Nb}Ka
Encrypted term(s) < Na *Nb > with key Ka is readable by recipient only.
The encrypted/fresh nonce Na has been received back in new component: {Na Nb}Ka
Outgoing test for A because fresh term Na was sent out earlier in < {A Na}Kb >
Sender may be attempting to initiate an outgoing test by transmitting Nb in encrypted form.

<Parties> : <Message> >> C -> A : {Na Nb}Ka
Encrypted term(s) < Na Nb > with key Ka is readable by recipient only.

<Parties> : <Message> >> B -> A : {B Na}Ka
Encrypted term(s) < B Na > with key Ka is readable by recipient only.

<Parties> : <Message> >> A -> B : {Na}Kb
Encrypted term(s) < Na > with key Kb is readable by recipient only.

<Parties> : <Message> >> A -> B : {Nb}Kb
Encrypted term(s) < Nb > with key Kb is readable by recipient only.
The encrypted/fresh nonce Nb has been received back in new component: {Nb}Kb
Outgoing test for B because fresh term Nb was sent out earlier in < {Na Nb}Ka >

<Parties> : <Message> >> A -> C : {Nb}Kc
Encrypted term(s) < Nb > with key Kc is readable by recipient only.

Figure 3.5 – Output of NS with duplicate transmission of nonce

We learn from this example that the SPA is not being fooled with repetitive transactions. However, it is catching and printing the existence of the two legitimate outgoing tests that occur. As one can see, there are numerous ways to arrange the order of these tests and the components within, but at this point we can be fairly certain the SPA is capable of finding the tests.

3.6 Summary

In conclusion, this chapter covered in detail the basic operation of the SPA. Numerous test cases and the Needham-Schroeder protocol were run and we use them to establish a baseline of reliability. Finally, there was a modification done to the Needham-Schroeder protocol to show that obvious mistakes in a transaction, whether intentional or not, could be captured. As a result of these tests, we've shown that the SPA is capable of detecting instances of incoming, outgoing and unsolicited authentication tests as well as basic improper events occurring during the course of a protocol run. We also introduce something termed outgoing/incoming test, which is the case where a received term meets criteria for both outgoing and incoming tests as described in chapter two. In the next chapter, we will expand upon this and run the SPA against numerous protocols that exhibit numerous characteristics.

IV. Analysis and Results

4.1 Chapter Overview

This chapter demonstrates the results of running the SPA on several regular protocol runs. These tests show if any authentications tests occur in the protocol and where. Following the discussion of normal protocol runs, several more 'incorrect', or seemingly incorrect, protocol runs are investigated. The purpose of creating incorrect protocols is to show that the SPA is capable of discovering erroneous scenarios that may occur in protocols. The means by which these particular scenarios are chosen is discussed later.

4.2 Results of Protocol Analyses

This section covers the overall results of running known protocols. As discussed earlier, protocols are grouped into categories. Categories can be broken up into protocols that use symmetric cryptography verse asymmetric cryptography. Another category is protocols which use trusted verse non-trusted third parties. Although there do exist hybrid protocols, the protocols chosen most assuredly fit into the above categories. The layout of the protocols is taken almost directly from [29].

In Chapter 3, outgoing, incoming and unsolicited tests output is shown in great detail. In this section, since testing occurs on large established protocols, previously unseen output is displayed. Although warnings and errors are self explanatory, there is another test condition that is introduced that may not be self-evident. The test is called a pseudo-unsolicited test. This test does not alter the definition of an unsolicited test in any way, but what it does do is show how one party is receiving a challenge from another that it

has previously sent a message to. In other words, because both parties understand the protocol, *A* may expect something back from *B*, in one case, the minimum reply might be *A*'s fresh nonce. However, simultaneously if *B* solicits a challenge of his own, *A* must now reply with the correct answer. The reason for the slight name change is based upon the generic nature of the analyzer. This analyzer is able to review any protocol without knowing how the protocol works. But because of this, it does have to keep track of intercommunicating parties and know who is sending what to whom but does not necessarily know that a reply or request is part of the protocol. Therefore, because a party has sent something to another means receiving something back from the party is not necessarily 'unsolicited'.

4.2.1 Wide Mouth Frog Protocol

The wide mouth frog protocol (Figure 4.1) involves the use of symmetric cryptography in conjunction with a trusted third party. In this protocol, the initiating principal generates a temporary session key, along with a timestamp. These are passed to the server, along with the identification of the party in which the initiating principal wishes to communicate. The server then passes the timestamp and session key onto the intended recipient.

The SPA recognizes all known tests. In the first message it correctly identifies the fact that *A* may be attempting to initiate two different outgoing tests. This does not mean they are completed; only that *A* is transmitting a legitimate test component that contains two fresh terms. *S*'s receipt of this valid test component with two fresh terms inside and the use of proper symmetric cryptography mean there exists two unsolicited tests from

S's perspective. In the second pass, *S* has created his own test component containing a fresh nonce and thus may be initiating his own outgoing test. From *B*'s point of view, the receipt of this test component with two fresh terms, one generated by *S* and the other by *A*, means *B* has two unsolicited tests. In this example, the SPA accurately detects all relevant tests that occur as shown in Figure 4.1.

A -> S : A {*Na1 B *Kab}Kas
S -> B : {*Ns1 A Kab}Kbs

<Parties> : <Message> >> A -> S : A {*Na1 B *Kab}Kas
Encrypt term(s) < *Na1 B *Kab > with key Kas is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.
Unsolicited test for S because of nonce Na1 within test component < {Na1 B Kab}Kas >
Sender may be attempting to initiate an outgoing test by transmitting Kab in encrypted form.
Unsolicited test for S because of nonce Kab within test component < {Na1 B Kab}Kas >

<Parties> : <Message> >> S -> B : {*Ns1 A Kab}Kbs
Encrypt term(s) < *Ns1 A Kab > with key Kbs is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Ns1 in encrypted form.
Unsolicited test for B because of nonce Ns1 within test component < {Ns1 A Kab}Kbs >
Unsolicited test for B because of nonce Kab within test component < {Ns1 A Kab}Kbs >

Figure 4.1 -Wide-Mouth Frog Protocol

4.2.2 Yahalom Protocol

In the Yahalom protocol, the SPA starts out by showing how principal *A* is initiating an incoming test by sending out a fresh term. The next message shows how *B* is initiating his own outgoing test and at the same by doing so, an unsolicited test occurs for the server, represented as *S*. Also, an unsolicited test occurs for nonce Na because it is still fresh from *S*'s view and it exists within a valid test component. The next step then shows how *A* gets its nonce, Na, back; thus completing the run of its incoming test. At the same time, it shows an unsolicited test for *A* because of a fresh term within a valid test component from *S*. In this pass, the first warning is generated. *S* has generated one

key and is transmitting it in two separate components. Because of this, if *S* were to get back one of these keys, it can not be certain from whom and therefore they are invalidated as a test component with regard to outgoing tests. The final message sees an unsolicited test for *B* as well as *B*'s receipt of his fresh nonce back completing a run of an outgoing/incoming test.

The significance of running Yahalom in the SPA is that the SPA is challenged with multiple parties, multiple test cases and an encounter with its first negated test component and warning message. The results of the SPA were expected so therefore we conclude that it has successfully completed analysis of this protocol.

A -> B : A *Na
B -> S : B {A Na *Nb}Kbs
S -> A : {B *Kab Na Nb}Kas {A *Kab}Kbs
A -> B : {A Kab}Kbs {Nb}Kab

<Parties> : <Message> >> A -> B : A *Na
Sender may be attempting to initiate an incoming test by transmitting Na in the clear.

<Parties> : <Message> >> B -> S : B {A Na *Nb}Kbs
Encrypt term(s) < A Na *Nb > with key Kbs is readable by both sender/receiver.
Unsolicited test for S because of nonce Na within test component < {A Na Nb}Kbs >
Sender may be attempting to initiate an outgoing test by transmitting Nb in encrypted form.
Unsolicited test for S because of nonce Nb within test component < {A Na Nb}Kbs >

<Parties> : <Message> >> S -> A : {B *Kab Na Nb}Kas {A *Kab}Kbs
Encrypt term(s) < B *Kab Na Nb > with key Kas is readable by both sender/receiver.
Unsolicited test for A because of nonce Kab within test component < {B Kab Na Nb}Kas >
Sender may be attempting to initiate an outgoing test by transmitting Kab in encrypted form.
The unencrypted/fresh nonce Na has been received back in new component: {B Kab Na Nb}Kas
Incoming test for A because fresh term Na was sent out earlier in < Na >
Unsolicited test for A because of nonce Nb within test component < {B Kab Na Nb}Kas >
Encrypted term(s) < A *Kab > with key Kbs is readable by sender only.
Warning: Sender is transmitting nonce Kab in two separate components, invalidating its use as an outgoing test.
Sender may be attempting to initiate an outgoing test by transmitting an invalid Kab in encrypted form.

<Parties> : <Message> >> A -> B : {A Kab}Kbs {Nb}Kab
Encrypted term(s) < A Kab > with key Kbs is readable by recipient only.
Unsolicited test for B because of nonce Kab within test component < {A Kab}Kbs >

Encrypt term(s) < Nb > with key Kab is readable by both sender/receiver.
The encrypted/fresh nonce Nb has been received back in new component: {Nb}Kab
Outgoing/Incoming test for B because fresh term Nb was sent out earlier in < {A Na Nb}Kbs >

Figure 4.2 – Yahalom Protocol

4.2.3 Woo-Lam Protocol

In this section, the Woo-Lam protocol is examined. From this point onward, rather than explain line by line each protocol, only key events of the protocol and the SPA's conclusions are discussed. In Woo-Lam, which is already identified to have a flaw [5], the SPA recognizes two tests. The first is the completed incoming test from *B*'s view. The second is the unsolicited test from the servers' point of view. However in Guttman's work, [5], he concludes that their does not exist a legitimate incoming test. This reasoning is based on the notion that another node could produce a received component of the same form. Since the SPA performs tests based on string comparisons, in a test like this, type comparisons would be more beneficial. Hence, a model checking language may prove more effective in this type of scenario.

A -> B : A
B -> A : *Nb1
A -> B : {Nb1}Kas
B -> S : {A {Nb1}Kas}Kbs
S -> B : {Nb1}Kbs

<Parties> : <Message> >> A -> B : A

<Parties> : <Message> >> B -> A : *Nb1
Sender may be attempting to initiate an incoming test by transmitting Nb1 in the clear.

<Parties> : <Message> >> A -> B : {Nb1}Kas
Encrypted term(s) < Nb1 > with key Kas is readable by sender only.

<Parties> : <Message> >> B -> S : {A {Nb1}Kas}Kbs
Encrypt term(s) < A {Nb1}Kas > with key Kbs is readable by both sender/receiver.
Encrypted term(s) < Nb1 > with key Kas is readable by recipient only.
Unsolicited test for S because of nonce Nb1 within test component < {Nb1}Kas >

<Parties> : <Message> >> S -> B : {Nb1}Kbs
Encrypt term(s) < Nb1 > with key Kbs is readable by both sender/receiver.
The unencrypted/fresh nonce Nb1 has been received back in new component: {Nb1}Kbs
Incoming test for B because fresh term Nb1 was sent out earlier in < Nb1 >

Figure 4.3 – Woo-Lam Protocol

4.2.4 Neuman-Stubblebine Protocol

The next protocol reviewed is the Neuman-Stubblebine protocol. In this test, two different scenarios are executed. The first (Figure 4.4) puts the * tag on the nonces and session keys generated by the server and the ticket generated by *B*. In other words, anything created fresh and unique is tagged in order to identify all potential tests. The second run (Figure 4.5) shows the analysis done without placing the * on the keys and tickets generated by the different parties. The reason for this is to show that in order to detect the presence of tests, it is important that the analyzers of the protocol know what they intend to use as unique/fresh terms, thus they have more flexibility when doing analysis.

In the first test, numerous examples of unsolicited tests are shown due to the sheer volume of freshly tagged terms. At this point, it is important to note that the SPA does not disqualify any components but does make note of the fact that in step 3 a fresh term (a key) is being transmitted in two components. Although this negates their validity as outgoing tests, the SPA still makes the correct assessment and determines that unsolicited tests involving these components do occur.

A -> B : A *Na
B -> S : B {A Na *tb}Kbs *Nb
S -> A : {B Na *Kab tb}Kas {A *Kab tb}Kbs Nb
A -> B : {A *Kab tb}Kbs {Nb}Kab

<Parties> : <Message> >> A -> B : A *Na

Sender may be attempting to initiate an incoming test by transmitting Na in the clear.

<Parties> : <Message> >> B -> S : B {A Na *tb}Kbs *Nb
Encrypt term(s) < A Na *tb > with key Kbs is readable by both sender/receiver.
Unsolicited test for S because of nonce Na within test component < {A Na tb}Kbs >
Sender may be attempting to initiate an outgoing test by transmitting tb in encrypted form.
Unsolicited test for S because of nonce tb within test component < {A Na tb}Kbs >
Sender may be attempting to initiate an incoming test by transmitting Nb in the clear.

<Parties> : <Message> >> S -> A : {B Na *Kab tb}Kas {A *Kab tb}Kbs Nb
Encrypt term(s) < B Na *Kab tb > with key Kas is readable by both sender/receiver.
The unencrypted/fresh nonce Na has been received back in new component: {B Na Kab tb}Kas
Incoming test for A because fresh term Na was sent out earlier in < Na >
Unsolicited test for A because of nonce Kab within test component < {B Na Kab tb}Kas >
Sender may be attempting to initiate an outgoing test by transmitting Kab in encrypted form.
Unsolicited test for A because of nonce tb within test component < {B Na Kab tb}Kas >
Encrypted term(s) < A *Kab tb > with key Kbs is readable by sender only.
Warning: Sender is transmitting nonce Kab in two separate components, invalidating its use as an outgoing test.
Sender may be attempting to initiate an outgoing test by transmitting an invalid Kab in encrypted form.

<Parties> : <Message> >> A -> B : {A *Kab tb}Kbs {Nb}Kab
Encrypted term(s) < A *Kab tb > with key Kbs is readable by recipient only.
Unsolicited test for B because of nonce Kab within test component < {A Kab tb}Kbs >
The encrypted/fresh nonce tb has been received back in new component: {A Kab tb}Kbs
Outgoing/Incoming test for B because fresh term tb was sent out earlier in < {A Na tb}Kbs >
Encrypt term(s) < Nb > with key Kab is readable by both sender/receiver.
The unencrypted/fresh nonce Nb has been received back in new component: {Nb}Kab
Incoming test for B because fresh term Nb was sent out earlier in < Nb >

Figure 4.4 – Neuman-Stubblebine with Tags on Keys/Tickets

In Figure 4.5, the Neuman-Stubblebine is run on the SPA without attaching the * to the keys or ticket. The only difference here is that it follows more closely with the results made by Guttman [5]. However, it is not incorrect to show a key or timestamp, generated similarly to a nonce, as fresh and denoting it with a fresh identifier as previously shown. Nor does it negate the conclusion of the previous example.

A -> B : A *Na1
B -> S : B {A *Na1 Tb}Kbs *Nb1
S -> A : {B Na1 Kab Tb}Kas {A Kab Tb}Kbs Nb1
A -> B : {A Kab Tb}Kbs {Nb1}Kab

<Parties> : <Message> >> A -> B : A *Na1
Sender may be attempting to initiate an incoming test by transmitting Na1 in the clear.

<Parties> : <Message> >> B -> S : B {A *Na1 Tb}Kbs *Nb1
Encrypt term(s) < A *Na1 Tb > with key Kbs is readable by both sender/receiver.
Sender is attempting to intiate a test with an old/invalidated nonce!
Unsolicited test for S because of nonce Na1 within test component < {A Na1 Tb}Kbs >
Sender may be attempting to initiate an incoming test by transmitting Nb1 in the clear.

<Parties> : <Message> >> S -> A : {B Na1 Kab Tb}Kas {A Kab Tb}Kbs Nb1
Encrypt term(s) < B Na1 Kab Tb > with key Kas is readable by both sender/receiver.
The unencrypted/fresh nonce Na1 has been received back in new component: {B Na1 Kab Tb}Kas
Incoming test for A because fresh term Na1 was sent out earlier in < Na1 >
Encrypted term(s) < A Kab Tb > with key Kbs is readable by sender only.

<Parties> : <Message> >> A -> B : {A Kab Tb}Kbs {Nb1}Kab
Encrypted term(s) < A Kab Tb > with key Kbs is readable by recipient only.
Encrypt term(s) < Nb1 > with key Kab is readable by both sender/receiver.
The unencrypted/fresh nonce Nb1 has been received back in new component: {Nb1}Kab
Incoming test for B because fresh term Nb1 was sent out earlier in < Nb1 >

Figure 4.5 – Neuman-Stubblebine w/o Tags on Keys/Tickets

4.2.5 Needham-Schroeder with Server

In this test, the Needham-Schroeder protocol is executed on the SPA again. However, unlike before, the server portion of the protocol is included (Figure 4.6). This protocol is drawn directly from [29]. However, for clarification, the keys Kas and Kbs are used to show when the server is communicating with *A* and *B* specifically. Normal operation of the protocol entails the server using its private key for encryption of these messages and assumes both parties have the servers' public key. The use of this key format does not alter the results of the test.

A -> S : A B
S -> A : {Kb B}KaS
A -> B : {A *Na}Kb
B -> S : B A
S -> B : {Ka A}KbS
B -> A : {Na *Nb}Ka
A -> B : {Nb}Kb

<Parties> : <Message> >> A -> S : A B

<Parties> : <Message> >> S -> A : {Kb B}KaS
Encrypt term(s) < Kb B > with key KaS is readable by both sender/receiver.

<Parties> : <Message> >> A -> B : {A *Na}Kb
Encrypted term(s) < A *Na > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Na in encrypted form.

<Parties> : <Message> >> B -> S : B A

<Parties> : <Message> >> S -> B : {Ka A}KbS
Encrypt term(s) < Ka A > with key KbS is readable by both sender/receiver.

<Parties> : <Message> >> B -> A : {Na *Nb}Ka
Encrypted term(s) < Na *Nb > with key Ka is readable by recipient only.
The encrypted/fresh nonce Na has been received back in new component: {Na Nb}Ka
Outgoing test for A because fresh term Na was sent out earlier in < {A Na}Kb >
Sender may be attempting to initiate an outgoing test by transmitting Nb in encrypted form.

<Parties> : <Message> >> A -> B : {Nb}Kb
Encrypted term(s) < Nb > with key Kb is readable by recipient only.
The encrypted/fresh nonce Nb has been received back in new component: {Nb}Kb
Outgoing test for B because fresh term Nb was sent out earlier in < {Na Nb}Ka > >

Figure 4.6 – Needham-Schroeder Protocol (w/ Server)

This test results in output similar to earlier tests with Needham-Schroeder. The only real difference is the interaction with the server that occurs between each party.

4.2.6 Kerberos Protocol

The Kerberos protocol results are very similar to Neuman-Stubblebine. In Chapter 2, an in-depth description of how this protocol works is given. The layout used here, which contains more specific information in each message, is taken directly from [29]. Regarding the analysis, there are no surprises in the output. The only subjective action was not to tag all keys and timestamps as fresh, even though they are. The reason is that, based on the Neuman-Stubblebine output, nothing further would be gained from this except a more lengthy output.

C -> A : U G L1 *N1
A -> C : U {U C G *Kcg Tst Tex}Kag {G *Kcg Tst Tex N1}Ku
C -> G : S L2 *N2 {U C G *Kcg Tst Tex}Kag {C *T1}Kcg
G -> C : U {U C S *Kcs Tst1 Tex1}Kcg {S *Kcs Tst1 Tex1 N2}Kcg
C -> S : {U C S Kcs Tst1 Tex1}Kcg {C *T2}Kcs
S -> C : {T2}Kcs

<Parties> : <Message> >> C -> A : U G L1 *N1
Sender may be attempting to initiate an incoming test by transmitting N1 in the clear.

<Parties> : <Message> >> A -> C : U {U C G *Kcg Tst Tex}Kag {G *Kcg Tst Tex N1}Ku
Encrypted term(s) < U C G *Kcg Tst Tex > with key Kag is readable by sender only.
Sender may be attempting to initiate an outgoing test by transmitting Kcg in encrypted form.
Encrypted term(s) < G *Kcg Tst Tex N1 > with key Ku is readable by neither sender nor receiver.
Warning: Sender is transmitting nonce Kcg in two separate components, invalidating its use as an outgoing test.
Sender may be attempting to initiate an outgoing test by transmitting an invalid Kcg in encrypted form.

<Parties> : <Message> >> C -> G : S L2 *N2 {U C G *Kcg Tst Tex}Kag {C *T1}Kcg
Sender may be attempting to initiate an incoming test by transmitting N2 in the clear.
Encrypted term(s) < U C G *Kcg Tst Tex > with key Kag is readable by recipient only.
Unsolicited test for G because of nonce Kcg within test component < {U C G Kcg Tst Tex}Kag >
Encrypt term(s) < C *T1 > with key Kcg is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting T1 in encrypted form.
Unsolicited test for G because of nonce T1 within test component < {C T1}Kcg >

<Parties> : <Message> >> G -> C : U {U C S *Kcs Tst1 Tex1}Kcg {S *Kcs Tst1 Tex1 N2}Kcg
Encrypt term(s) < U C S *Kcs Tst1 Tex1 > with key Kcg is readable by both sender/receiver.
Pseudo-unsolicited test for C because Kcs is a newly received fresh nonce, but C has sent items to G previously.
Sender may be attempting to initiate an outgoing test by transmitting Kcs in encrypted form.
Encrypt term(s) < S *Kcs Tst1 Tex1 N2 > with key Kcg is readable by both sender/receiver.
Warning: C is seeing Kcs again...but it's tagged with * (fresh) identifier!
Warning: Sender is transmitting nonce Kcs in two separate components, invalidating its use as an outgoing test.
Sender may be attempting to initiate an outgoing test by transmitting an invalid Kcs in encrypted form.
The unencrypted/fresh nonce N2 has been received back in new component: {S Kcs Tst1 Tex1 N2}Kcg
Incoming test for C because fresh term N2 was sent out earlier in < N2 >

<Parties> : <Message> >> C -> S : {U C S Kcs Tst1 Tex1}Kcg {C *T2}Kcs
Encrypted term(s) < U C S Kcs Tst1 Tex1 > with key Kcg is readable by sender only.
Encrypt term(s) < C *T2 > with key Kcs is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting T2 in encrypted form.
Unsolicited test for S because of nonce T2 within test component < {C T2}Kcs >

<Parties> : <Message> >> S -> C : {T2}Kcs
Encrypt term(s) < T2 > with key Kcs is readable by both sender/receiver.
The encrypted/fresh nonce T2 has been received back in new component: {T2}Kcs
Outgoing/Incoming test for C because fresh term T2 was sent out earlier in < {C T2}Kcs >

Figure 4.7 – Kerberos Protocol

4.2.7 Analyzing Otway-Rees

In this section, the Otway-Rees protocol is analyzed using the SPA. Figure 4.8 shows the output generated from the SPA.

A -> B : *M A B {*Na1 *M A B}Kas
B -> S : M A B {Na1 M A B}Kas {*Nb1 M A B}Kbs
S -> B : M {Na1 *Kab}Kas {Nb1 *Kab}Kbs
B -> A : M {Na1 Kab}Kas

<Parties> : <Message> >> A -> B : *M A B {*Na1 *M A B}Kas
Sender may be attempting to initiate an incoming test by transmitting M in the clear.
Encrypted term(s) < *Na1 *M A B > with key Kas is readable by sender only.
Sender may be attempting to initiate an outgoing test by transmitting Na1 in encrypted form.
Sender may be attempting to initiate an outgoing test by transmitting M in encrypted form.
Sender is sending the same nonce M out both encrypted and plain.
It may only be used to complete an incoming test.

<Parties> : <Message> >> B -> S : M A B {Na1 M A B}Kas {*Nb1 M A B}Kbs
Encrypted term(s) < Na1 M A B > with key Kas is readable by recipient only.
Unsolicited test for S because of nonce Na1 within test component < {Na1 M A B}Kas >
Unsolicited test for S because of nonce M within test component < {Na1 M A B}Kas >
Encrypt term(s) < *Nb1 M A B > with key Kbs is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Nb1 in encrypted form.
Unsolicited test for S because of nonce Nb1 within test component < {Nb1 M A B}Kbs >

<Parties> : <Message> >> S -> B : M {Na1 *Kab}Kas {Nb1 *Kab}Kbs
Encrypted term(s) < Na1 *Kab > with key Kas is readable by sender only.
Sender may be attempting to initiate an outgoing test by transmitting Kab in encrypted form.
Encrypt term(s) < Nb1 *Kab > with key Kbs is readable by both sender/receiver.
The encrypted/fresh nonce Nb1 has been received back in new component: {Nb1 Kab}Kbs
Outgoing/Incoming test for B because fresh term Nb1 was sent out earlier in < {Nb1 M A B}Kbs >
Pseudo-unsolicited test for B because Kab is a newly received fresh nonce, but B has sent items to S previously.
Warning: Sender is transmitting nonce Kab in two separate components, invalidating its use as an outgoing test.
Sender may be attempting to initiate an outgoing test by transmitting an invalid Kab in encrypted form.

<Parties> : <Message> >> B -> A : M {Na1 Kab}Kas
Encrypted term(s) < Na1 Kab > with key Kas is readable by recipient only.
The encrypted/fresh nonce Na1 has been received back in new component: {Na1 Kab}Kas
Outgoing/Incoming test for A because fresh term Na1 was sent out earlier in < {Na1 M A B}Kas >
Unsolicited test for A because of nonce Kab within test component < {Na1 Kab}Kas >

Figure 4.8 – Output of Otway-Rees protocol run

In the Otway-Rees run there is nothing new being seen by the analyzer that has not already been demonstrated. The only important thing to get from this is that the SPA is capable of accurately finding authentication tests and examples of basic mischievous/erroneous activity.

4.3 Non-Regular Protocol Test Cases

This section examines several test cases meant to test the reliability of the SPA against cases that, although not representative of any real protocol, may undoubtedly surface in a similar form. These test cases are essentially designed to test the analyzers ability to recognize common mischievous activities as defined in the Dolev-Yao threat model.

4.3.1 Repeating Message

This section examines how the SPA can verify a case in which a principal retransmits an identical message back to the sender. In Figure 4.9, principal *A* transmits a test component containing a fresh nonce. From a receiver's point of view, this constitutes an unsolicited test. However, the intended recipient simply retransmits the message back to the sender. Because the component received back is not altered, *A* cannot deduce that an intended party performed any action on the component; therefore, this does not constitute an outgoing test.

A -> B : {*Na}Kab
B -> A : {Na}Kab

<Parties> : <Message> >> A -> B : {*Na}Kab
Encrypt term(s) < *Na > with key Kab is readable by both sender/receiver.
Sender may be attempting to initiate an outgoing test by transmitting Na in encrypted form.

Unsolicited test for B because of nonce Na within test component < {Na}Kab >

<Parties> : <Message> >> B -> A : {Na}Kab
Encrypt term(s) < Na > with key Kab is readable by both sender/receiver.
The encrypted nonce Na has been received back, but in a duplicated transmission!
The component {Na}Kab was sent out identically by this sender.

Figure 4.9 - Repeating Message

4.3.2 Sub Term Relationship of Encrypted Test Component

This section covers a test that demonstrates the analyzers ability to verify usage of test components as a whole in an improper manner. Recall from Chapter 2 that the definition of a test component stipulates that a test component cannot exist as a proper subterm of any other term on any other regular node. In this example, *B* attempts to use *A*'s encrypted message as his own 'fresh' term when communicating with *C*. The analyzer, which looks ahead, realizes this and flags the original component as invalid! Because of this, *A* is no longer able to complete is outgoing test. Since we are using asymmetric cryptography, no unsolicited tests occur either. The only test that might have occurred in this run would be an outgoing if *C* gets his fresh nonce back at a later time.

A -> B : {*Na}Kb
B -> C : {{Na}Kb}Kc
C -> B : {{Na}Kb *Nc}Kb
B -> A : {Na Nc}Ka

<Parties> : <Message> >> A -> B : {*Na}Kb
Encrypted term(s) < *Na > with key Kb is readable by recipient only.
{Na}Kb is an invalid test component because it's either a proper
subterm of another component or is a duplicate transmission!
Receiver is receiving an invalid test component {Na}Kb otherwise, it would be an unsolicited test with fresh term Na

<Parties> : <Message> >> B -> C : {{Na}Kb}Kc
Encrypted term(s) < {Na}Kb > with key Kc is readable by recipient only.
Encrypted term(s) < Na > with key Kb is readable by sender only.

<Parties> : <Message> >> C -> B : {{Na}Kb *Nc}Kb
Encrypted term(s) < {Na}Kb *Nc > with key Kb is readable by recipient only.

Encrypted term(s) < Na > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Nc in encrypted form.

<Parties> : <Message> >> B -> A : {Na Nc}Ka
Encrypted term(s) < Na Nc > with key Ka is readable by recipient only.

Figure 4.10 - Sub-Term Re-Encryption

4.3.3 Unreadable Encryption on Messages

The final improper protocol test involves testing the SPA's ability to check for proper encryption. In Figure 4.11, the SPA is running the Needham-Schroeder protocol but on line two of the input file, the encryption ensures *A* cannot read the message. Therefore, the SPA should not show any example of an outgoing test. Correctly so, the SPA does not. In fact, notice the SPA's ability to demonstrate duplicate transmission of messages.

A -> B : {*Na}Kb
B -> A : {Na *Nb}Kb
A -> B : {Na Nb}Kb

<Parties> : <Message> >> A -> B : {*Na}Kb
Encrypted term(s) < *Na > with key Kb is readable by recipient only.
Sender may be attempting to initiate an outgoing test by transmitting Na in encrypted form.

<Parties> : <Message> >> B -> A : {Na *Nb}Kb
Encrypted term(s) < Na *Nb > with key Kb is readable by sender only.
Sender may be attempting to initiate an outgoing test by transmitting Nb in encrypted form.

<Parties> : <Message> >> A -> B : {Na Nb}Kb
Encrypted term(s) < Na Nb > with key Kb is readable by recipient only.
The encrypted nonce Nb has been received back, but in a duplicated transmission!
The component {Na Nb}Kb was sent out identically by this sender.

Figure 4.11 – Needham-Schroeder with improper encryption

4.4 Summary

This chapter described the results of running several normal protocols and several incorrect protocols on the SPA. The output of each test is described in detail in order for the reader to have a greater appreciation of what the tool is doing. It is evident that the

tool provides a reliable means of determining if and where any authentication tests occur in the occurrence of a particular protocol. The SPA also demonstrates its ability to ascertain improper behavior, as demonstrated in the 'improper' protocols. Although the SPA is not designed to show weaknesses in a protocol, the absence of authentication tests should raise flags regarding a protocol and thus increase skepticism regarding its security. The next chapter gives final analysis of the results of running the SPA on communication protocols.

V. Conclusions and Recommendations

5.1 Chapter Overview

This chapter provides conclusions as to the success of the research and provides a roadmap for future work in the field of protocol analysis. It provides insight into what the Secure Protocol Analyzer accomplishes and how another modeling language may or may not provide better analysis of protocols in the future.

5.1 Conclusions of Research

The SPA successfully shows that protocol analysis with a tool developed in Java is highly valuable in the performance of protocol analysis. In particular, the SPA is able to determine when and where outgoing, incoming and unsolicited tests occur within a protocol run. Using string comparisons vice type comparisons requires specific values be given and does limit the application to analysis based on completed static runs. However, putting together numerous protocols in generic text files proves much easier than individual protocol development as noted in other protocol analyzers [27, 11]. It also allows for much quicker analysis of the protocol because it does not have to dynamically create a search tree, instead it only examines the post-run state of the protocol as entered in the input text file. In conclusion, the SPA allows the taking of any protocol as input in a standard text file and generates accurate output that shows occurrences of authentication tests, and it does so very quickly.

5.3 Significance of Research

This research is significant because it creates a simple-to-use tool that effectively shows the presence of authentication tests. Although there are numerous other methods for analyzing protocols, this method proves easy to use, accurate in its results and, more importantly, it enables any protocol to be entered in with minimal effort by the user. Another important result of this work is that output from the SPA can be tailored to one's unique work. This opens the possibility that output from the SPA could be used as input for other theorem checking tools or protocol analysis tools that may look for different aspects regarding a particular protocol. Finally, this fast automation of analysis is important because the longer a protocol is left in use without in-depth analysis being performed on it, the more the chance of mischievous persons finding a potential weakness during its use and exploiting it.

5.4 Recommendations for Future Research

Although the tool is fairly complete in its present form, added functionality will certainly improve the effectiveness of the tool. For example, adding functionality that would include provisions for penetrator capabilities. This would entail adding in a means to examine static runs and evaluate what information a penetrator is able to derive based on the Dolev-Yao threat model. From this information, the tool might be able to start developing its own messages in an effort to show the user that it is possible to implant false messages leading to potential havoc on the principals. Another function worth adding is that of showing all the keys used during the protocol exchange. From this, the analyzer can show the keys actually used for encryption, whether they are safe or not,

then show the set assumed to be safe as defined by Guttman. Also, as stated in [5], since the authentication tests work even if n and n' are on different strands, add to the analyzer the ability to find the tests within different strands.

Another course of action is development in another modeling language altogether. Using an analyzer developed through a model checker would allow dynamic creation of search trees where numerous different states can be analyzed showing not only where and when authentication tests occur but also how to better develop the protocol to ensure the authentication tests do occur and that certain states are never reached.

The other main recommendation for future research is to produce a similar analyzer using the Maude language specifically. As stated earlier, dynamic evaluation of protocols allows users to see first-hand the actual operation of a protocol run, versus the after effect of the run as entered in static format. As explained in Chapter 2, Maude's rewriting ability would enable it to generate dynamic states very easily. With this dynamic evaluation, using a Dolev-Yao threat model, also built into the modeling language, the user can ascertain all the possible states that a protocol can be in and all the possible information that can be derived. This includes states that the protocol should not necessarily be in. Limitations would have to be placed on this method to ensure that state space explosion and other problems do not occur as described in greater detail in Chapter 2.

Although there are numerous other protocol analyzers [11, 12, 24], none seem to exhibit the all-knowing power to ascertain whether protocols are susceptible to attack. However, built upon the foundation of a solid, powerful and easy to use tool the research

and tool generated here gives analysts a clear roadmap into the future of protocol analysis.

Appendix A – Security Protocol Analyzer Code

```
/*
 * Parser.java
 *
 * Created on December 8, 2003
 */

package protocolB.Parser;

import java.io.*;
import java.text.*;
import protocolB.*;

/**
 * This class parses text from a Reader into a Protocol.
 * @author  rgraham
 */
public class Parser
{
    /** Creates a new instance of Parser */
    public Parser(Reader in) {
        this.in = in;
    }

    /** The lexical analyzer, which returns the next token from the
Reader. */
    private Token nextToken() throws ParseException {
        StringBuffer sb = new StringBuffer();
        int type;
        int ch;         // It is so annoying that read returns int!

        if (nextToken != null)
            return nextToken;

        // Get the next input character
        ch = getChar();

        // Skip white space
        while (ch == ' ' || ch == '\t' || ch == '\r')
            ch = getChar();

        // Read until one complete token is found
        if (ch == -1)
            type = Token.EOF;
        else if (ch == '\n')
            type = Token.EOL;
        else if (ch == '{')
            type = Token.LBRACE;
        else if (ch == '}')
            type = Token.RBRACE;
        else if (ch == ':')
            type = Token.COLON;
        else if (ch == '*')
            type = Token.STAR;
```

```
        else if (ch == '-') {
            getChar();        // Skip '>' without checking it
            type = Token.ARROW;
        }
        else if (Character.isLetterOrDigit((char) ch)) {
            do {
                sb.append((char) ch);
                ch = getChar();
            } while (Character.isLetterOrDigit((char) ch));
            prev = ch;
            type = Token.IDENTIFIER;
        }
        else
            throw new ParseException("Unrecognized character '" + ch +
"'", offset);

        nextToken = new Token(type, sb.toString());
        return nextToken;
    }

    /** Reads the next logical character from the input, which may be
the
     *  readahead character.
     */
    private int getChar() {
        int ch = -1;

        if (prev != 0) {
            ch = prev;
            prev = 0;
        }
        else {
            try {
                ch = in.read();
            } catch (Exception e) {
                System.err.println(e);
                System.err.println("Aborting.");
                System.exit(1);
            }
            offset++;
        }

        return ch;
    }

    public Protocol parse() throws ParseException {
        Protocol p = new Protocol();
        Message m;

        while (nextToken().getType() != Token.EOF) {
            try {
                m = parseMessage();
                p.addMessage(m);
            } catch (ParseException e) {
```

```
                System.err.println(e + " at offset " +
e.getErrorOffset());
                // Skip tokens until EOL or EOF
                while (nextToken().getType() != Token.EOL
                        && nextToken().getType() != Token.EOF)
                    nextToken = null;
            }
        }
        Match(Token.EOF);

        return p;
    }

    /** Matches a specified token type against the next input token and
throws an
     *  exception if it doesn't match.
     */
    protected Token Match(int t) throws ParseException {
        Token next = nextToken();

        if (next.getType() != t)
            throw new ParseException("Match Bad token " +
Token.names[next.getType()]
                                     + ", expected " + Token.names[t],
offset);
        nextToken = null;  // Consume the token
        return next;
    }

    protected Encryption parseEncryption() throws ParseException {
        Term m;
        Text key;

        Match(Token.LBRACE);
        m = parseTerm();
        Match(Token.RBRACE);
        key = parseText();

        return new Encryption(m, key);
    }

    protected Text parseText() throws ParseException {
        boolean fresh = false;
        Token next = nextToken();

        if (next.getType() == Token.STAR) {
            fresh = true;
            Match(Token.STAR);
        }
        Token id = Match(Token.IDENTIFIER);
        return new Text(id.getText(), fresh);
    }

    protected Message parseMessage() throws ParseException {
        Text from, to;
```

```
        Term m;

        from = parseText();
        Match(Token.ARROW);
        to = parseText();
        Match(Token.COLON);
        m = parseTerm();
        Match(Token.EOL);
        return new Message(from.getText(), to.getText(), m);
    }

    /** Parses a sequence of one or more Terms.  If more than one, it
returns
     *  a Sequence containing the Terms found.
     */
    public Term parseTerm() throws ParseException {
        Term term = null, nextTerm;
        Token next = nextToken();

        while (next.getType() != Token.RBRACE && next.getType() !=
Token.EOL
               && next.getType() != Token.EOF) {
            if (next.getType() == Token.LBRACE)
                nextTerm = parseEncryption();
            else if (next.getType() == Token.IDENTIFIER
                     || next.getType() == Token.STAR)
                nextTerm = parseText();
            else
                throw new ParseException("Unexpected token "
                                         + Token.names[next.getType()]
                                         + ", expected LBRACE,
IDENTIFIER or STAR",
                                         offset);

            if (term == null)
                term = nextTerm;
            else if (term instanceof Sequence)
                ((Sequence) term).addTerm(nextTerm);
            else {
                Sequence temp = new Sequence();
                temp.addTerm(term);
                temp.addTerm(nextTerm);
                term = temp;
            }
            next = nextToken();
        }

        if (term == null)
            throw new ParseException("Term expected", offset);

        return term;
    }

    protected Reader in;
    private int prev = 0;              // Lookahead character
```

```
    private Token nextToken = null;     // Lookahead token
    private int offset = 0;
}

/*
 * Token.java
 *
 * Created on December 8, 2003
 */

package protocolB.Parser;

/**
 * This class represents a single Token parsed from an input stream.
 *
 * @author  rgraham
 */
public class Token {

    /** Creates a new instance of Token */
    public Token(int type, String text) {
        this.type = type;
        this.text = text;
    }

    public int getType() {
        return type;
    }

    public String getText() {
        return text;
    }

    public static final int EOF = 0;
    public static final int LBRACE = 1;
    public static final int RBRACE = 2;
    public static final int IDENTIFIER = 3;
    public static final int ARROW = 4;
    public static final int COLON = 5;
    public static final int EOL = 6;
    public static final int STAR = 7;

    public static String names[] = { "EOF", "LBRACE", "RBRACE",
"IDENTIFIER",
                                     "ARROW", "COLON", "EOL", "STAR" };

    protected int type;
    protected String text;
}

package protocolB;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 24 Feb 2004
```

```
This is just the abstract class that encryption, sequence and term are
extending
*/
import java.util.*;
public abstract class Term
{
  public Term()
  { }
  public abstract void getReadableTexts(String lfrom, String lto,
Vector p,
  boolean readable, boolean sent_plain, String comp,
  Vector list_of_components, Vector old_sent, Vector bad_test_comp,
Vector temp_holdings, Vector text_obj, String type_enc);
  //IF there is a better way to do this I just don't kow how right
now...but at
  //least I haven't reached 13 arguments, 12 means perfection -> 13
means insanity!!!
  public abstract void getComp(Vector bad_test_comp,Vector
temp_holdings, String lfrom, String lto, boolean outside);//Go through
tree and get {h}k
  public abstract void getTextObj(Vector text_obj);

}

package protocolB;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 24 Feb 2004
RGraham says:
 * A Text is a primitive (atomic) Term.  Common texts are principal
names, keys
 * and nonces.  A fresh text is one that is generated dynamically in
such a way
 * as to be unique among all other texts in use (at least to high
probability,
 * as by a pseudo-random process with a sufficient text length).
 *
What more can I say?  However, these is where principal.addnonce is
called for each individual instance of a term.
It then goes into that particular parties instance and checks their
vectors to see if it is completing a test
*/
import java.util.*;
public class Text extends Term
{
    /** Creates a new instance of Text */
    public Text(String text, boolean fresh) {
        this.text = text;
        this.fresh = fresh;
        if (fresh) this.wasfresh = true; //if it is fresh it will have
once been fresh
        //this wasfresh should never change because it only says the
term was fresh at one time
    }
```

```
    public String getText() {
        return text;
    }

    public boolean isFresh() {
        return fresh;
    }

    public void getReadableTexts(String lfrom, String lto, Vector p,
boolean readable, boolean sent_plain, String comp,
                                  Vector list_of_components, Vector
old_sent, Vector bad_test_comp, Vector temp_holdings, Vector text_obj,
String type_enc)
    {
      int y = 0;

      if (this.isFresh())
      {
        while (y < p.size())
        {
            Principal P = (Principal) p.elementAt(y);
            if (P.lname.equalsIgnoreCase(lfrom))
            P.addnonce(text, "sender", lfrom, lto, readable,
sent_plain, true, comp, list_of_components, old_sent, bad_test_comp,
temp_holdings, text_obj, type_enc);

            if (P.lname.equalsIgnoreCase(lto))
            P.addnonce(text, "receiver", lfrom, lto, readable,
sent_plain, true, comp, list_of_components, old_sent,
bad_test_comp,temp_holdings, text_obj, type_enc);
            y++;
        }
      }//End of if it's fresh
      else//Even if it's not fresh, if it was fresh the text_obj vector
has each text object in it, that way I can get more about this
particular text
      {
        int U = 0;
        String temp = "";
        while (U < p.size())
        {
           Principal P = (Principal) p.elementAt(U);

           if (P.lname.equalsIgnoreCase(lto))//lto (below) used to be
temp, not sure why?should have used comments earlier!
           P.addnonce(text, "receiver", lfrom, lto, readable,
sent_plain, false, comp, list_of_components, old_sent, bad_test_comp,
temp_holdings, text_obj, type_enc);

           U++;
        }
      }//End of else it's not
    }
```

```
    public String toString() {
        return (fresh ? "*" : "") + text;
    }

    public void getComp(Vector bad_test_comp, Vector temp_holdings,
String lfrom, String lto, boolean outside)//, Vector text_obj)
    {    }//End of get comp

    public void getTextObj(Vector text_obj)
    {
      //I just want my text object which is the {}K stuff, in principal
I'll check if my
      //individual term is in the bad 't'
      boolean toadd = false;//once I loop through vector if text is not
in there this changes to true
      int t = 0;              //Then it gets added to the list, ensures
duplicates aren't added

      while (t < text_obj.size())
      {
        Text temptext = (Text) text_obj.elementAt(t);
        if (temptext.getText().equalsIgnoreCase(this.getText()))
            {  toadd = true;   }
        t++;
      }//end of while
      if (!toadd)
          {  text_obj.addElement(this);    }
      toadd = false;//change it back regardless...even though when I
come back it gets reset anyways!!!
    }//End of getTextObj

    protected String text;
    protected boolean fresh;
    protected boolean is_nonce;
    protected boolean wasfresh;
}

package protocolB;
import java.util.*;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 24 Feb 2004

 * A Sequence is a concatenation of Terms.  Concatenation is assumed to
be
 * associative.
 *  This class calls getreadabletext based on type of comp it is
working with...
 * @author  rgraham
 */
public class Sequence extends Term
{
    /** Creates a new instance of Sequence */
    public Sequence() {
```

```
        terms = new ArrayList();
    }

    public void addTerm(Term t) {
        terms.add(t);
    }

    public ListIterator getTerms() {
        return terms.listIterator();
    }

    public void getComp(Vector bad_test_comp,Vector temp_holdings,
String lfrom, String lto, boolean outside)
    {
      System.out.println("Enter getComp in sequ");
      for (ListIterator i = terms.listIterator(); i.hasNext(); )
      {
        Term u = (Term) i.next();
        System.out.println(u.toString());
        if (u instanceof Sequence)
        {//Once I am inside encryption, it has its own way of handling
sequences...slightly different from sequence method
          u.getComp(bad_test_comp, temp_holdings,lfrom, lto, outside);
        }
        else if (u instanceof Encryption)//This won't be internal
encryption, that is handled inside encrpytion
        {//This is the case where the message comes in for example:  A
{a}K<-then it will call encryption to check inside this
            temp_holdings.addElement(u.toString());//should add outside
encryption to temp_holdings

            //////////
                 int max = u.toString().length();
             int Y = 0;
             String hold = "";
             while (Y < max)
             {
               if (!u.toString().substring(Y,
Y+1).equalsIgnoreCase("*"))
               hold = hold + u.toString().substring(Y, Y+1);
               Y++;
             }
             temp_holdings.addElement(hold);

             String hold1 = hold + lfrom;
             String hold2 = hold + lto;
             if (!temp_holdings.contains(hold1))
                if (!temp_holdings.contains(hold2))
                temp_holdings.addElement(hold1);
            //////////

            u.getComp(bad_test_comp,temp_holdings, lfrom, lto,
outside);//Now let's go inside the encryption and check for term types
        }
```

```
        else //What else could it be?//it must be text don't really
care, it does nothing anyways
            u.getComp(bad_test_comp,temp_holdings, lfrom, lto,
outside);
      }
    }//End of get comp function

    public void getTextObj(Vector text_obj)
    {
      for (ListIterator i = terms.listIterator(); i.hasNext(); )
      {
        Term u = (Term) i.next();
        u.getTextObj(text_obj);//Call whatever instance of term the
i.next is
      }
    }

    public void getReadableTexts(String lfrom, String lto, Vector p,
boolean readable,
                                 boolean sent_plain, String comp,
Vector list_of_components,
                                 Vector old_sent, Vector bad_test_comp,
Vector temp_holdings, Vector text_obj, String type_enc)
    {
      for (ListIterator i = terms.listIterator(); i.hasNext(); )
      {

       Term u = (Term) i.next();

       if (u instanceof Sequence)//Could be more sequence of stuff
       {    u.getReadableTexts(lfrom, lto, p, readable, sent_plain,
comp, list_of_components, old_sent, bad_test_comp, temp_holdings,
text_obj, type_enc);  }

       else if (u instanceof Encryption)
       {
       String temporary = u.toString();
       u.getReadableTexts(lfrom, lto, p, readable, sent_plain,
temporary, list_of_components, old_sent, bad_test_comp, temp_holdings,
text_obj, type_enc);
       list_of_components.addElement(temporary);
       }       //Work with component then add it to "I've seen it
already" vector

       else //It's either encryption or text object...
       {//don't add text to component list....
       u.getReadableTexts(lfrom, lto, p, readable, sent_plain, comp,
list_of_components, old_sent, bad_test_comp, temp_holdings, text_obj,
type_enc);
       }//End of else it isn't sequence

      }//End of for loop
    }//End of GetReadable Texts() function

    public String toString() {
```

```
        StringBuffer sb = new StringBuffer();

        for (ListIterator i = terms.listIterator(); i.hasNext(); ) {
            sb.append(((Term) i.next()).toString());
            if (i.hasNext())
                sb.append(" ");
        }

        return sb.toString();
    }

    protected List terms;
}

package protocolB;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 23 Feb 2004

This is where the main() function is.  This is the driver of the
program!  Here is where the following occurs:
individual instances of principal are created from the 'from' and 'to'
of the message, they are placed in a vector
and passed around.  Also, each message is broken into components and
analyzed if it's a duplicate or subterm (subterm
verification also is checked in encryption class).  Depending on check,
the comp is placed in wither test_comp vector
or bad_comp vector.   After this steps are done, it then starts the
process by analyzing 1 message at a time, depending
on what the instance is, it call getreadabletext.  Since a message may
contain sub-parts, getreadabletext is also called
recursively in other classes such as encryption and text.
*/
//import Parser.*;
import protocolB.Parser.*;
import java.io.*;
import java.util.*;
import java.text.ParseException;

public class ProtocolAnalyzer {

     public static void main(String[] args) {
        boolean err = false;
        Protocol protocol = null;
        Reader reader = null;

        Vector num_parties = new Vector();
        Vector parties = new Vector();
        Vector list_of_components = new Vector(); //Lists all
componentsfu
        Vector old_sent = new Vector();

        Vector bad_test_comp = new Vector(); //this one only worries
about
```

```
        //whether some t = {h}k is not a proper subcomponent of any
regular node

        Vector temp_holdings = new Vector();//This started out as a
temp item but now has evolved into a real vector to be used
        //it contains the encrypted terms that can act as legitimate
test components...

        Vector text_obj = new Vector(); //Make an instance of each text
and pass it around

        //The wonderful world of non-GUI main screens
        System.out.println();
        System.out.println(">>>>>>>>>>>>>> Protocol-Analyzer Version
1.2 <<<<<<<<<<<<<");

        if (args.length == 0) {
            System.out.println("Nothing to parse.");
            System.exit(1);
        }

        System.out.println("Parsing from file '" + args[0] + "'");
        try {
            reader = new FileReader(args[0]);
        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }

        Parser p = new Parser(reader);
        try {
            protocol = p.parse();
        } catch (ParseException e) {
            System.out.println(e + " at offset " + e.getErrorOffset());
            err = true;
        } catch (Exception e) {
            System.out.println(e);
            err = true;
        }
        if (!err)
            System.out.println("File Contents:\n" + protocol);

        for (ListIterator i = protocol.listIterator(); i.hasNext(); )
        {
          Message t = (Message) i.next();
          //Lets count the number of unique parties in the given
protocol
          if (!num_parties.contains(t.from))
num_parties.addElement(t.from);
          if (!num_parties.contains(t.to))
num_parties.addElement(t.to);
        }//End of for loop which creates principal instances...

          //Start creating instances of party for each player in the
protocol
```

```
          String temp = "";
          int j = 0;
          while (j<num_parties.size())
          {
            temp = num_parties.elementAt(j).toString();
            Principal PPP = new Principal(temp);
            parties.addElement(PPP);
            temp = "";
            j++;
          }

///////////This section will run through the whole protocol and get all
components for
///////////later testing on subterm relationships...
///////////Will also do the text_obj stuff in here
          for (ListIterator i = protocol.listIterator(); i.hasNext(); )
          {
            Message t = (Message) i.next();
            Term g = (Term) t.getTerm();

          if (g instanceof Sequence)
          {
             Sequence S = (Sequence) g;

             for (ListIterator k = S.getTerms(); k.hasNext();)
             {
              Term l = (Term) k.next();
              l.getComp(bad_test_comp, temp_holdings, t.from, t.to,
true);
              l.getTextObj(text_obj);
             }//End of for loop within sequence
           }//End of if statement
           else if (g instanceof Encryption)
           {
             temp_holdings.addElement(g);
             //This part makes copy and puts it in without * so I can
check for duplicate later
             int max = g.toString().length();
             int Y = 0;
             String hold = "";
             while (Y < max)
             {
               if (!g.toString().substring(Y,
Y+1).equalsIgnoreCase("*"))
               hold = hold + g.toString().substring(Y, Y+1);
               Y++;
             }
             temp_holdings.addElement(hold);

             String hold1 = hold + t.from;
             String hold2 = hold + t.to;
             if (!temp_holdings.contains(hold1))
                if (!temp_holdings.contains(hold2))
                temp_holdings.addElement(hold1);
```

```
            g.getComp(bad_test_comp, temp_holdings, t.from, t.to,
true);
            g.getTextObj(text_obj);
          }
          else if (g instanceof Text)
          {
            g.getComp(bad_test_comp, temp_holdings, t.from, t.to,
true);
            g.getTextObj(text_obj);
          }
         }//End of for loop through evaluation of entire protocol for
potential test components

      //What this part does is take the terms in my array of suspect
comp's and remove *, so that
      //someone doesn't retransmit exact same message back
      //for example: {*N}Kmn  then resend as {N}Kmn is not valid
outoging test!
  /*        int h = 0;
         int max =  bad_test_comp.size();
         String holder = "";
         String temporary2 = "";
         while (h < max)
         {
           holder = bad_test_comp.elementAt(h).toString();
           int y = 0;
           while (y <holder.length())
           {
               if (!holder.substring(y,y+1).equalsIgnoreCase("*"))
                   temporary2 = temporary2 + holder.substring(y,y+1);
                   y++;
           }//End of innner while

           bad_test_comp.addElement(temporary2);
           temporary2 = "";
           h++;
         }//End of outer while*/

         //Take out the * in the temp holding vector
      /*    int h = 0;
         int max =  temp_holdings.size();
         String holder = "";
         String temporary2 = "";
         while (h < max)
         {
           holder = temp_holdings.elementAt(h).toString();
           int y = 0;
           while (y <holder.length())
           {
               if (!holder.substring(y,y+1).equalsIgnoreCase("*"))
                   temporary2 = temporary2 + holder.substring(y,y+1);
                   y++;
           }//End of innner while

           temp_holdings.addElement(temporary2);
```

```
            temporary2 = "";
            h++;
          }//End of outer while*/

///MORE DEBUG CODE  JUST DUMPS VECTORS OF COMP DEBUG CODE///
     /*     int K = 0;
          while (K < bad_test_comp.size())
          {//This is for testing purposes
           System.out.println("Bad test comp: " +
bad_test_comp.elementAt(K).toString());
            K++;
          }

          K = 0;
          while (K < temp_holdings.size())
          {//This is for testing purposes
           System.out.println("Temp holdings: " +
temp_holdings.elementAt(K).toString());
            K++;
          }*/

//DEBUG CODE DEBUG CODE //Test the text_obj vector //DEBUG CODE
     /*     int w = 0;
          while (w < text_obj.size())
          {
            Text temptext = (Text) text_obj.elementAt(w);
            System.out.println("The PA output in vector is: " +
temptext.toString());
            w++;
          }//end of while*/
////////////End of my latest monster code which follows well that of
the pasta class! Still,
////////////it is quite ingenious....or I must be losing it, or I am a
genious...only voice number 3
////////////in my head knows the truth  :-)

        for (ListIterator i = protocol.listIterator(); i.hasNext(); )
        {
          Message t = (Message) i.next();
          System.out.println();
          System.out.println("<Parties> : <Message> >> " +
t.toString());
          Term g = (Term) t.getTerm();
          String type_enc = "ASYM";

          if (g instanceof Sequence)
          {
            Sequence S = (Sequence) g;

            for (ListIterator k = S.getTerms(); k.hasNext();)
            {
              Term l = (Term) k.next();
              l.getReadableTexts(t.from, t.to, parties, false, true,
l.toString(), list_of_components, old_sent, bad_test_comp,
temp_holdings, text_obj, type_enc);
```

```
            }
          }
          else if (g instanceof Encryption)
          {
          g.getReadableTexts(t.from, t.to, parties, false, true,
g.toString(), list_of_components, old_sent, bad_test_comp,
temp_holdings, text_obj, type_enc);
          list_of_components.addElement(g.toString());//Work with
component then add it to "I've seen it already" vector
          }
          else
          {
          g.getReadableTexts(t.from, t.to, parties, false, true,
g.toString(), list_of_components, old_sent, bad_test_comp,
temp_holdings, text_obj, type_enc);
          }//Don't add plain text to comp vector
        }

    }//End of main
}//End of class
package protocolB;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 23 Feb 2004

This is a key class.  Here is where an addnonce, that is called from
text class, ends up.  This has
numerous flags and variables passed in.  This class does the test for
tests.  Whenever the string value
is set to sender, values sent in apply to that particular instance of
sender, created in protocolanalyzer class.
Likewise for receiver.  Not much else to say on this class except that
tests work basically like this:  Whenever I
am in sender mode, whatever nonce is passed in is placed in a vector
(sent_nonces if encr/sent_unencrypted is sent unencr)
As long as the comp isn't disqualified, that is.  These flags are
checked mostly in all the if's of the receiver.
Then whenever I am receiver, I check through my sent_nonce and
sent_unencerypted vectors and see if the nonce passed
in is there, if so it completes whatever test is applicable.

*/
import java.util.*;
public class Principal
{
        String lname = "";
        String the_fresh_nonce = "";
        boolean lwasfresh = false;
        Vector Sent_nonces = new Vector();
        Vector old_nonces = new Vector();
        Vector Sent_unencrypted = new Vector();
        Vector people_I_sent_stuff_to = new Vector();
        Vector messages_I_saw = new Vector();
        Vector sent_message_plain = new Vector();
```

```
        Vector nonces_I_saw = new Vector();
        Vector comp_I_sent = new Vector();
        Text hold_text;

        public Principal(String tname)
        {    lname = tname;     }

        public void addnonce(String nonce, String person, String from,
String lto, boolean readable, boolean sent_plain, boolean is_fresh,
            String comp, Vector list_of_components, Vector old_sent,
Vector bad_test_comp, Vector temp_holdings, Vector text_obj, String
type_enc)
        {//readable is set in encryption class...isfresh is set in text
class....
          //sent_plain is set to ture initially, then changed in
encryption class
          //from and lto only have values depending on whose key it is

           String temp_comp = comp + lto;//This is how I set the string
to look for duplicate retransmission of messages
           //in other words, if I sent out lets say A sent out a comp
{a}Ka and it is concat with A so it is {a}Ka + A = {a}KaA  then later I
am receiving it back
           //to check for duplication I concatenate the lto,
           //let's say it is A getting it back, so then it becomes
{a}KaA, well they match so obviously someone sent A's original comp
back <-that's bad!

          //This part goes and get's the object of the nonce and then
we can ascertain it's current status (fresh or was fresh)
          //that way if I want to add more info about a text object, I
can and it's real easy to get now!
          //Maybe I can add a value that says who the originator was,
then I can populate that value here locally
          //and then based on lfrom and lto I can decide more
accurately about the tests? FUTURE WORK!
          int U = 0;
          while (U < text_obj.size())
          {
            Text temptext = (Text) text_obj.elementAt(U);
            if (nonce.equalsIgnoreCase(temptext.getText()))
            {//Could populate any variable I want here that the
particular text object might hold. (Future work?)
              //or I could just stop the loop and work with temptext
              hold_text = temptext; //I may need to use this object
later..in fact I will!!!
              lwasfresh = temptext.wasfresh;
            }//End of if and end of populating variables!
            U++;
          }//End of while through text objects


          //Let's go ahead and take out the * in the comp because a
principal may try to resend it later after
          //having read it and thus taking it's freshenss out...
```

```
          int B = 0;
          String placeholder = "";
          while (B < comp.length())
          {
           if (!comp.substring(B,B+1).equalsIgnoreCase("*"))
            placeholder =  placeholder + comp.substring(B,B+1);
            B++;
          }
          comp =  placeholder;
            //Start the principal anaylsis from the sender's view
            if (person.equals("sender"))
            {

              if (Sent_nonces.contains(nonce))
              {
                int e = Sent_nonces.indexOf(nonce);
                String old_comp = (String) Sent_nonces.elementAt(e +
1);
                //if (!comp.equalsIgnoreCase(old_comp))
               // if (temp_holdings.contains(comp) &&
temp_holdings.contains(old_comp) && comp.equalsIgnoreCase(old_comp))
               if (old_comp.indexOf(comp) > 0 | comp.indexOf(old_comp)
> 0)
                {}
               else
                {
                  System.out.println("Warning: Sender is transmitting
nonce " + nonce + " in two separate components, invalidating its use as
an outgoing test.");
                  Sent_nonces.removeElementAt(e + 1);
                  Sent_nonces.removeElement(nonce);
                  nonce = "an invalid " + nonce;
                }
              }//End of if sent_plain has it and I am resending in
another comp

              if (sent_plain && !nonces_I_saw.contains(nonce))
              {
                //since these nonces are sent plain I have to go
through vector of invalid test comp
                //and see if later this nonce doesn't show up there,
thus negating it's validity
                int e = 0;
                System.out.println("Sender may be attempting to
initiate an incoming test by transmitting " + nonce + " in the
clear.");
                sent_message_plain.addElement(nonce);
                sent_message_plain.addElement(comp);
              }//End of initiating incoming test

              //Start of initiating outgoing test
              else if ((!bad_test_comp.contains(comp) ||
!temp_holdings.contains(comp)) && !nonces_I_saw.contains(nonce))
              {//If the test compopnent is in bad test comp and temp
holdings it can't be used
```

```
               if ((bad_test_comp.contains(comp) &&
temp_holdings.contains(comp)))
                  { }
                  else
                  {
                System.out.println("Sender may be attempting to
initiate an outgoing test by transmitting " + nonce + " in encrypted
form. ");
                Sent_nonces.addElement(nonce);
                Sent_nonces.addElement(comp);
                  }
              }
                  ///End of initiating outgoing test

              else if (temp_holdings.contains(comp) &&
bad_test_comp.contains(comp))
              {
                System.out.println(comp + " is an invalid test
component because it's either a proper");
                System.out.println("subterm of another component or is
a duplicate transmission!");
              }

              else if (nonces_I_saw.contains(nonce))
                System.out.println("Sender is attempting to intiate a
test with an old/invalidated nonce!!!");

              if (readable)
              {  people_I_sent_stuff_to.addElement(lto);  }//Can the
person I send it to read it?
              //If the recipient can't read it then it won't count
later when checking for unsolicited

              if (old_nonces.contains(nonce))//Stops reuse of nonces
and won't allow tests to work
              {   int z = old_nonces.indexOf(nonce);
                     z = z + 1;
                     System.out.println("I've sent nonce " + nonce + "
out before in message " + old_nonces.elementAt(z));

Sent_nonces.removeElement(old_nonces.elementAt(z)); //Get rid of
message associated with it
                     Sent_nonces.removeElement(nonce);//Get rid of
previously sent out nonce
                     if (sent_message_plain.contains(nonce))
                        {
                          z = sent_message_plain.indexOf(nonce);

sent_message_plain.removeElementAt(z + 1);
                          sent_message_plain.removeElement(nonce);
                         }
               }

            if (is_fresh)//Because nonces may come in bundles only get
the fresh one for display
```

```
            {    the_fresh_nonce = nonce;     }
            else
              the_fresh_nonce = "";

              if (!from.equalsIgnoreCase(""))
              {
              String temporary2 = "";
              int h = 0;
              while (h < comp.length())
              {
                if
(!comp.substring(h,h+1).equalsIgnoreCase("*"))//duplicate!  See below
note:
                    temporary2 = temporary2 +
comp.substring(h,h+1);//duplicate, not needed but not sure top part
will stay
                    h++;//so not going to remove this yet!!!
              }
                //comp_I_sent.addElement(temporary2);//putting comp in
old vector of stuff I already sent
              }

              //In case the person sends out a nonce both encrypted and
plain I will remove them and announce failed test...
          if (sent_message_plain.contains(nonce) &&
Sent_nonces.contains(nonce))
            {
                System.out.println("Sender is sending the same nonce "
+ nonce + " out both encrypted and plain.");
                System.out.println("It may only be used to complete an
incoming test.");
                //remove the bad nonce from both respective vectors...
                  //  int z = sent_message_plain.indexOf(nonce);
                 //   sent_message_plain.removeElementAt(z + 1);
                //    sent_message_plain.removeElement(nonce);
                    int z =  Sent_nonces.indexOf(nonce);
                    Sent_nonces.removeElementAt(z + 1);
                    Sent_nonces.removeElement(nonce);
            //now we put this comp in the bad test comp vector so we
don't flag an unsolicited test
                  //  bad_test_comp.addElement(comp);
                    nonces_I_saw.addElement(nonce);
            }

            }//End of person being the sender//////////////////

            if (person.equals("receiver"))//////Start the receiver
            {
                        if (nonces_I_saw.contains(nonce) && is_fresh)
                { System.out.println("Warning: " + this.lname + " is
seeing " + nonce + " again...but it's tagged with * (fresh)
identifier!");   }

                        if (sent_plain &&  is_fresh)
                  nonces_I_saw.addElement(nonce);
```

```
                                                    // &&
!bad_test_comp.contains(comp) this was part of restrictions on
unsolicited test...
                if (!this.people_I_sent_stuff_to.contains(from)
                    && !Sent_nonces.contains(nonce) &&
!messages_I_saw.contains(comp)
                    && readable && (is_fresh | lwasfresh) &&
!nonces_I_saw.contains(nonce)
                    && !sent_message_plain.contains(nonce) &&
!old_sent.contains(nonce) && !type_enc.equalsIgnoreCase("ASYM"))
                {
                System.out.print("Unsolicited test for " + this.lname);
                System.out.println(" because of nonce " + nonce + "
within test component < " + comp + " > ");
                nonces_I_saw.addElement(nonce);
                }//End of If for unsolicited test

                else if (!this.people_I_sent_stuff_to.contains(from) &&
!bad_test_comp.contains(comp)
                     && !messages_I_saw.contains(comp) && readable &&
is_fresh && (nonces_I_saw.contains(nonce)
                      | sent_message_plain.contains(nonce) |
Sent_nonces.contains(nonce)))
                    { System.out.println("Sender may be attempting to
initiate a test with an old nonce!"); }

                else if (!this.people_I_sent_stuff_to.contains(from) &&
bad_test_comp.contains(comp) &&
                    !messages_I_saw.contains(comp) && readable &&
is_fresh)
                {
                  System.out.print("Receiver is receiving an invalid
test component " + comp);
                  System.out.println(" otherwise, it would be an
unsolicited test with fresh term " + nonce);
                }

                if (this.people_I_sent_stuff_to.contains(from) &&
readable && is_fresh
                    && !nonces_I_saw.contains(nonce) &&
!Sent_nonces.contains(nonce)
                    && !sent_message_plain.contains(nonce) &&
!old_sent.contains(nonce) && !type_enc.equalsIgnoreCase("ASYM"))
                {
                  System.out.println("Pseudo-unsolicited test for "+
this.lname  + " because " + nonce
                  + " is a newly received fresh nonce, but " +
this.lname + " has sent items to " + from + " previously.");
                  nonces_I_saw.addElement(nonce);
                }
                else if (this.people_I_sent_stuff_to.contains(from) &&
readable && is_fresh
                    && !nonces_I_saw.contains(nonce) &&
Sent_nonces.contains(nonce))
```

```
                { System.out.println("Sender may be attempting to
intitiate a test with an old nonce!"); }


                if (comp_I_sent.contains(comp) && !sent_plain)
                {
                    System.out.println("Evaluation of term " + nonce  +
" indicates it is part of a component " + comp);
                    System.out.println("which is being retransmitted.
Therefore, test component " + comp + " is invalidated!");
                }

                  if (!comp_I_sent.contains(comp) &&
!old_nonces.contains(nonce))
                {//If it is fresh and not a duplicate test comp and not
a proper subterm somewhere

                  if (Sent_nonces.contains(nonce) && sent_plain)
                  {
                   int t = Sent_nonces.indexOf(nonce);
                   t = t + 1;
                   System.out.println("The encrypted/fresh nonce " +
nonce + " has been received back in new component: " + comp);
                   System.out.print("Outgoing test for " + this.lname +
" because fresh term " + nonce + " was sent out earlier");
                           System.out.println(" in < " +
Sent_nonces.elementAt(t)+ " > ");
                   old_nonces.addElement(nonce);//Now I've got it back
I add nonce to old nonces vector

old_nonces.addElement(Sent_nonces.elementAt(t));//add associated
message
                  }
                  if (sent_message_plain.contains(nonce) && !sent_plain
&&
                    !(bad_test_comp.contains(comp) &&
temp_holdings.contains(comp))&& !type_enc.equalsIgnoreCase("ASYM"))
                  {
                   int t = sent_message_plain.indexOf(nonce);
                   t = t + 1;
                           System.out.println("The unencrypted/fresh
nonce " + nonce + " has been received back in new component: " + comp);
                   System.out.print("Incoming test for " + this.lname +
" because fresh term " + nonce + " was sent out earlier");
                           System.out.println(" in < " +
sent_message_plain.elementAt(t) + " >");
                   old_nonces.addElement(nonce);

old_nonces.addElement(sent_message_plain.elementAt(t));
                  }//End of else for incoming
                  if (Sent_nonces.contains(nonce) && !sent_plain  &&
!temp_holdings.contains(temp_comp)&& type_enc.equalsIgnoreCase("ASYM"))
                  {
                   int t = Sent_nonces.indexOf(nonce);
                   t = t + 1;
```

```
                    System.out.println("The encrypted/fresh nonce " +
nonce + " has been received back in new component: " + comp);
                    System.out.print("Outgoing test for " + this.lname +
" because fresh term " + nonce + " was sent out earlier");
                              System.out.println(" in < " +
Sent_nonces.elementAt(t)+ " > ");
                    old_nonces.addElement(nonce);//Now I've got it back
I add nonce to old nonces vector

old_nonces.addElement(Sent_nonces.elementAt(t));//add associated
message
                   }//End of else if for outgoing test
                   else if (Sent_nonces.contains(nonce) && !sent_plain
&& !temp_holdings.contains(temp_comp))
                   {
                    int t = Sent_nonces.indexOf(nonce);
                    t = t + 1;
                    System.out.println("The encrypted/fresh nonce " +
nonce + " has been received back in new component: " + comp);
                    System.out.print("Outgoing/Incoming test for " +
this.lname + " because fresh term " + nonce + " was sent out earlier");
                              System.out.println(" in < " +
Sent_nonces.elementAt(t)+ " > ");
                    old_nonces.addElement(nonce);//Now I've got it back
I add nonce to old nonces vector

old_nonces.addElement(Sent_nonces.elementAt(t));//add associated
message
                   }//End of else if for outgoing/incoming test

                   if (temp_holdings.contains(temp_comp) &&
Sent_nonces.contains(nonce))
                   {
                     System.out.println("The encrypted nonce "+ nonce +
" has been received back, but in a duplicated transmission!");
                     System.out.println("The component " + comp + " was
sent out identically by this sender.");
                   }//End of retransmitted test
                 }
            }//End of receiver part of addnonce function
        }  //End of addnonce
}//End of class Principal
package protocolB;
/**
* Stephen Mancini and Robert Graham
* AFIT/ENG
* 23 Feb 2004

The purpose of this class is as follows:
Whenever an instance of an encrypted term is called, it happens here.
This class will ascertain if
a particular message is readable by sender, receiver, both or neither.
This happens through recursive calls
of getreadabletext() which is abstract and located several classes.
The arguments are the same, however, based
```

```
on who can read it, the from and to strings are set to relevant
setting.  For instance, if a sender can read it
the from string is set to that value.
Bottom line, based on key values, sender and receiver are set and
getreadabletext function is called...
*/
import java.util.*;

public class Encryption extends Term
{
    /** Creates a new instance of Encryption */
    public Encryption(Term t, Text k) {
        this.term = t;
        this.key = k;
    }

    public Term getTerm() {
        return term;
    }

    public Text getKey() {
        return key;
    }


    public void getComp(Vector bad_test_comp, Vector temp_holdings,
String lfrom, String lto, boolean outside)
    {//I only want to add those cases where t = {h}k so add encrypted
wihtin encrypted components!

      if (outside)
      {
        //This part is for whenever I might be bringing encryption from
the min part but lets say it is in a sequence
        //I still need to store that outside part and unfortunately
this can't be done in PA because it call getComp
        //of whatever object type it finds. So if outside is only
encryption I can handle it in PA otherwise I go inside
        //each object type's getComp and do whatever, in some cases
this misght perform different functions
        //so you may see repeating code ut chances are it is augmenting
something unique to that inside instance
        temp_holdings.addElement(this);
        int max = this.toString().length();
        int Y = 0;
        String hold = "";
        while (Y < max)
        {
          if (!this.toString().substring(Y, Y+1).equalsIgnoreCase("*"))
            hold = hold + this.toString().substring(Y, Y+1);
          Y++;
        }//End of while loop through string
        temp_holdings.addElement(hold);
        String hold1 = hold + lfrom;
        String hold2 = hold + lto;
```

```
        if (!temp_holdings.contains(hold1))
          if (!temp_holdings.contains(hold2))
                temp_holdings.addElement(hold1);
      }//End of adding component based on the fact that it is outside
but iin sequential order

      if (this.getTerm() instanceof Sequence)
      {
        Sequence S = (Sequence) this.getTerm();

        for (ListIterator k = S.getTerms(); k.hasNext();)
        {
          Term l = (Term) k.next();
          if (l instanceof Encryption)
          {//Not seeing the other term as encryption...
             if (temp_holdings.contains(l.toString()))
                  bad_test_comp.addElement(l);//encryption within
encryption...ruled out as test component

                  ///////////////Just removing any astrerisks
             int max = l.toString().length();
             int Y = 0;
             String hold = "";
             while (Y < max)
             {
               if (!l.toString().substring(Y,
Y+1).equalsIgnoreCase("*"))
               hold = hold + l.toString().substring(Y, Y+1);
               Y++;
             }
             bad_test_comp.addElement(hold);
             /////////////End of removing any asterisks

             l.getComp(bad_test_comp,temp_holdings, lfrom, lto,
false);//Make sure their isn't more encryption within
           }
          else
          {//not encr5yption?  Then either more sequence or text!  Love
abstract classes.
           l.getComp(bad_test_comp, temp_holdings,lfrom, lto, false);
          }
        }//End of for loop through sequence iterator
      }//End of if it's a sequence..if not sequence or encryption don't
worry about it, it must be text only.

      else if (this.getTerm() instanceof Encryption)//encryption within
encryption
        {//Any encrypted components wihthin encrypted components are
ruled out as test components
            if (temp_holdings.contains(this.getTerm().toString()))
            bad_test_comp.addElement(this.getTerm());
        }
    }//end of getComp() function
```

```
    public void getTextObj(Vector text_obj)
    {//Let us get all the text objects that are within encryped terms
      if (this.getTerm() instanceof Sequence)
      {
      Sequence s = (Sequence) this.getTerm();
      for (ListIterator i = s.getTerms(); i.hasNext(); )
      {
        Term u = (Term) i.next();
        u.getTextObj(text_obj);
      }
      }//end of sequence
      else if (this.getTerm() instanceof Text)
        this.getTerm().getTextObj(text_obj);
      else
        this.getTerm().getTextObj(text_obj);
    }//End of getTextObj()



    public void getReadableTexts(String lfrom, String lto, Vector p,
boolean readable,
                                  boolean sent_plain, String comp,
Vector list_of_components,
                                  Vector old_sent, Vector bad_test_comp,
Vector temp_holdings, Vector text_obj, String type_enc)
        {
      String temp = "";//If I can't open it I need to send a blank
recipient
      String temp_from = "";//I will set this based on whether or not
component is new regardless of encryption
      //Making assumotion that if comp is new that is where it
originates.
      String temp1 = this.key.toString().substring(1,2);//Set the first
value for the key
      String temp2 = "";//Maybe there are 2 values for the key...
      type_enc = "ASYM";  //What type of encryption?  Needed for
negating unsolicited tests

      if (comp.equalsIgnoreCase(""))
          comp = this.term.toString();//If it wasn't a sequence in
P..A... then just use this term

      if (this.key.toString().length()>2)
      {
        temp2 = this.key.toString().substring(2,3);
            type_enc = "SYM";
      }

      if (!list_of_components.contains(comp))
      {
        temp_from = lfrom;
        list_of_components.addElement(comp);//It's not getting added in
PA so this is a quick fix...
```

```
        //besides, most concern in protocols comes inside the encrypted
portions transmission.
      }

      if ((temp1.equalsIgnoreCase(lto) | temp2.equalsIgnoreCase(lto))
&& (temp1.equalsIgnoreCase(lfrom) | temp2.equalsIgnoreCase(lfrom)))
         {//sender and recipient can read it
         System.out.println("Encrypt term(s) < " + this.term + " > with
key " + this.getKey().toString() + " is readable by both
sender/receiver." );
         term.getReadableTexts(lfrom, lto, p, true, false, comp,
list_of_components, old_sent, bad_test_comp, temp_holdings, text_obj,
type_enc);// this.term.toString());
            }
       else if (temp1.equalsIgnoreCase(lto) |
temp2.equalsIgnoreCase(lto))
        {//Only the recipient can read it
      System.out.println("Encrypted term(s) < " + this.term + " > with
key " + this.getKey().toString() +" is readable by recipient only." );
        term.getReadableTexts(temp_from, lto, p, true, false,comp,
list_of_components, old_sent, bad_test_comp, temp_holdings, text_obj,
type_enc);//  this.term.toString());
        }
      else if (temp1.equalsIgnoreCase(lfrom) |
temp2.equalsIgnoreCase(lfrom))
        {//Only the sender can read it
      System.out.println("Encrypted term(s) < " + this.term + " > with
key " + this.getKey().toString() + " is readable by sender only.");
        term.getReadableTexts(lfrom, temp, p, false, false,comp,
list_of_components, old_sent, bad_test_comp, temp_holdings, text_obj,
type_enc);//  this.term.toString());
        }
      else
       {//Neither can read it
        System.out.println("Encrypted term(s) < " + this.term + " >
with key " + this.getKey().toString() + " is readable by neither sender
nor receiver.");
        term.getReadableTexts(temp_from, temp, p, true, false, comp,
list_of_components, old_sent, bad_test_comp, temp_holdings, text_obj,
type_enc);//If the key doen't match either party
       }
    }

    public String toString() {
        return "{" + term.toString() + "}" + key.toString();
    }

    protected Term term;
    protected Text key;
    protected boolean within = false;
}

package protocolB;

/**
```

```
 *
 * A Protocol is a sequence of {@link Message}s.
 * Created on December 8, 2003, 4:57 PM
 * @author  rgraham
 */
import java.util.*;
public class Protocol
{
    /** Creates a new instance of a Protocol. */
    public Protocol() {
        messages = new ArrayList();
    }

    public void addMessage(Message m) {
        messages.add(m);
    }

    public ListIterator listIterator() {
        return messages.listIterator();
    }

    public String toString() {
        StringBuffer sb = new StringBuffer();

        for (ListIterator i = messages.listIterator(); i.hasNext(); )
            sb.append(((Message) i.next()).toString() + "\n");

        return sb.toString();
    }

    protected List messages;

    private static class Class1 {
    }
}

package protocolB;

/**
 * A message is a term that one principal sends to another.
 *
 * @author  rgraham
 */
public class Message {

    /** Creates a new instance of Message */
    public Message(String from, String to, Term term) {
        this.from = from;
        this.to = to;
        this.term = term;
    }

    public String getReceiver() {
        return to;
    }
```

```
    public String getSender() {
        return from;
    }

    public Term getTerm() {
        return term;
    }

    public void setReceiver(String to) {
        this.to = to;
    }

    public void setSender(String from) {
        this.from = from;
    }

    public void setTerm(Term term) {
        this.term = term;
    }

    public String toString() {
        return from + " -> " + to + " : " + term;
    }

    protected String from;
    protected Term   term;
    protected String to;
}

/*
 * TermParser.java
 *
 * Created on December 8, 2003
 */

package protocolB;

import protocolB.Parser.*;
import java.io.*;
import java.text.ParseException;

/**
 * A TermParser tests the parser's ability to parse Terms.  The term to
parse is
 * specified on the command line (if it contains space, put it in
quotes).
 *
 * @author  rgraham
 */
public class TermParser
{
    public static void main(String[] args) {
        boolean err = false;
        Term term = null;
```

```
        if (args.length == 0) {
            System.out.println("Nothing to parse.");
            System.exit(1);
        }

        System.out.println("Parsing term: '" + args[0] + "'");
        Parser p = new Parser(new StringReader(args[0]));
        try {
            term = p.parseTerm();
        } catch (ParseException e) {
            System.out.println(e + " at offset " + e.getErrorOffset());
            err = true;
        } catch (Exception e) {
            System.out.println(e);
            err = true;
        }

        if (!err)
            System.out.println("Successful parse of '" + term + "'");
    }
}
```

Bibliography

1. Blanchet, Bruno. From Secrecy to Authenticity in Security Protocols. http://citeseer.nj.nec.com/correct/575612, 2000.

2. Guttman, Joshua D, Herzog Jonathon C. and F. Javier Fabrega. Strand Spaces: Proving Security Protocols Correct, Journal of Computer Security, 1999.

3. Guttman, Joshua D. and F. Javier Fabrega. Paths through Well-Behaved Bundles, (Abstract) June 2000.

4. Guttman, Joshua D. Security Protocol Design via Authentication Tests, Computer Security Foundations Workshop, April 2002.

5. Guttman, Joshua D. and F. Javier Fabrega. Authentication Tests, Proceedings 2000 IEEE Symposium on Security and Privacy, 2000.

6. Guttman, Joshua D. Key Compromise, Strand Spaces and the Authentication Tests, Electronic Notes in Theoretical Computer Science: 47, May 2001.

7. Guttman, Joshua D. Security Goals: Packet Trajectories and Strand Spaces, September 2000, Unpublished.

8. Guttman, Joshua D. and F. Javier Fabrega. Authentication Tests and the Structure of Bundles, November 2000.

9. Lowe, Gavin. Breaking and Fixing the Needham-Schroeder Public Key Protocol using FDR, Springer-Verlag 1996.

10. Saidi, Hassen. Towards Automatic Synthesis of Security Protocols, 2002.

11. Song, Dawn Wiaodong. Athena: A New Efficient Automatic Checker for Security Protocol Analysis, 1999.

12. Song, Dawn, Sergey Berezin and Adrian Perrig. Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis (Abstract).

13. Steiner, Jennifer G, Clifford Neuman and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems, March 1988.

14. Tung, Brian. The Moron's Guide to Kerberos: Version 1.2.2. December 1996.

15. http://www.w3.org/TR/SOAP, May 2003.

16. http://www.cs.cornell.edu/jif/, June 2003.

17. http://Maude.cs.uiuc.edu/overview.html, June 2003.

18. http://www.computer.org/proceedings/csfw/0201/02010192abs.htm, June 2003.

19. Clavel, Manuel and others. Maude 2.0 Manual. June 2003.

20. http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.htm , December 2003.

21. http://cliki.tunes.org/Maude, December 2003.

22. Cervesato, Iliano and others. A Comparison between Strand Spaces and Multiset Rewriting for Security Protocol Analysis, 2000.

23. http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?pi-calculus, November 2003.

24. Needham, Roger and Michael Schroeder. Using Encryption for Authentication in Large Networks of Computers, December 1978.

25. http://www.usingcsp.com/, December 2003.

26. Mao, Wenbo. A Structured Operational Modelling of the Dolev-Yao Threat Model. Hewlett-Packard Co. 2002.

27. Denker, G. and J. Mesegeur. Protocol Specification and Analysis in Maude, June 1998.

28. Song, D. and Adrian Perrig. Looking for Diamonds in the desert – Extending Protocol Generation to Three Party Authentication and Key Agreement Protocols. July, 2000.

29. Clark, John and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0. 17 November 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
23-03-2004	Master's Thesis	March 2003 – March 2004

4. TITLE AND SUBTITLE

AUTOMATING SECURITY PROTOCOL ANALYSIS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR (S)

Mancini, Stephen W., 1Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/04-12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Security Agency
Attn: Sylvan Pinsky (pinsky@thematrix.ncsc.mil)/ I333
9800 Savage Road
Ft George G. Meade, MD 207500-6704

10. SPONSOR/MONITOR'S ACRONYM(S)

NSA/I333

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT When Roger Needham and Michael Schroeder first introduced a seemingly secure protocol [24], it took over 18 years to discover that even with the most secure encryption, the conversations using this protocol were still subject to penetration. To date, there is still no one protocol that is accepted for universal use. Because of this, analysis of the protocol outside the encryption is becoming more important. Recent work by Joshua Guttman and others [9] have identified several properties that good protocols often exhibit. Termed "Authentication Tests", these properties have been very useful in examining protocols. The purpose of this research is to automate these tests and thus help expedite the analysis of both existing and future protocols.

The success of this research is shown through rapid analysis of numerous protocols for the existence of authentication tests. The result of this is that an analyst is now able to ascertain in near real-time whether or not a proposed protocol is of a sound design or whether an existing protocol may contain previously unknown weaknesses. The other achievement of this research is the generality of the input process involved. Although there exist other protocol analyzers, their use is limited primarily due to their complexity of use. With the tool generated here, an analyst needs only to enter their protocol into a standard text file; and almost immediately, the analyzer determines the existence of the authentication tests.

15. SUBJECT TERMS

Cryptography, Secure Communications

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Graham, Robert P., Maj, USAF
U	U	U	UU	103	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-6565 ext. 4715 (Robert.graham@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18