Modular Design for Robust Authentication Protocols

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Hilary A. Pike, B.S.

* * * * *

The Ohio State University

2004

Master's Examination Committee:

Paul A. G. Sivilotti, Adviser

Dong Xuan

Approved by

____

Adviser

Department of Computer and Information Science

© Copyright by

Hilary A. Pike

2004

ABSTRACT

Networks of computers use authentication protocols to verify their identities to one another. Some protocols, such as Kerberos, are used in a wide variety of environments with varying network timing parameters. Currently, the correctness of the Kerberos protocol is coupled to specific assumptions about synchrony. Thus, the correctness of this protocol must be verified on a case by case basis of deployment.

We present design techniques for constructing modular protocols based on abstract detection oracles. These oracles encapsulate timing parameters to decouple protocol correctness from explicit timing parameters. We demonstrate this approach by applying it to the Kerberos authentication protocol, and show how strengthening this oracle creates an augmented Kerberos protocol capable of tolerating sophisticated attacks.

To my husband, Scott M. Pike, for long hours of support, guidance, laughter, and challenging conversation.

ACKNOWLEDGMENTS

Many thanks to my colleagues, family, and friends for their support along this journey. I am extremely grateful to my adviser Paul Sivilotti who has provided unwaning patience, motivation, and support. Without his guidance, this thesis would be simply a drifting thought. Thanks also to Dong Xuan for participating in my thesis examination.

Thanks to Jason Hallstrom and Nigamanth Sridhar for many intriguing conversations that helped me clarify my research and explore new areas of computer science. A special thanks to the members of the Reusable Software Research Group (RSRG), Bruce Weide, Neelam Soundarajan, Tim Long, Bill Ogden, Paolo Bucci, and Bob Mathis, for helping me to refine each step of my research. Thanks to everyone in the Software Engineering Lab and Distributed Systems Seminar who made endless hours of work challenging and enjoyable.

Heartfelt thanks to my parents, Anne and Mark Stock, who consistently challenged me to see my true potential and follow my dreams. Finally, tremendous thanks to my sister, Leslie Snavely, who fills so many gaps, answers and asks the right questions, and helps me find satisfaction in all parts of life.

VITA

December 8, 1978	Born - Columbus, Ohio
2002	B.S. Computer Science and Engineering, The Ohio State University
2002-2003	Graduate Research and Teaching Associate, Computer and Information Science, The Ohio State University

FIELDS OF STUDY

Major Field: Computer and Information Science

TABLE OF CONTENTS

Page

Abstract ii

Dedication iii

Acknowledgments iv

Vita v

List of Figures viii

Chapters:

1. Introduction 1
   - 1.1 Problem 1
   - 1.2 Approach 2
   - 1.3 Thesis 2
   - 1.4 Organization 2

2. Background 4
   - 2.1 Authentication 4
   - 2.2 Abstract Protocol Notation 5
     - 2.2.1 Processes 5
     - 2.2.2 Security in Abstract Protocol Notation 8
   - 2.3 Kerberos 10
     - 2.3.1 Simple Exchange Protocol 13
     - 2.3.2 Additional Protection 16
   - 2.4 Suppress-Replay 23

3. Modeling Kerberos . . . 25
3.1 Kerberos in Abstract Protocol (AP) Notation . . . 25
3.1.1 States of Kerberos . . . 30
3.1.2 Correctness . . . 36
3.2 Oracle . . . 37
4. Kerberos with Intruders . . . 42
4.1 Intrusion States . . . 45
4.2 Vulnerability of Kerberos . . . 50
4.3 The Window of Vulnerability . . . 52
4.4 Modifying the Oracle . . . 55
4.5 Correctness . . . 56
5. Related Work . . . 57
6. Conclusions . . . 59
Appendices:
A. A Realization for Random without $\perp$ . . . 61
B. States . . . 62
B.1 Safe States . . . 62
B.2 Safe and Unsafe States . . . 64
Bibliography . . . 71

LIST OF FIGURES

Figure		Page
2.1	Simple Exchange: Initial Exchange of Messages	12
2.2	Simple Exchange: Final Exchange of Messages	14
2.3	Simple Exchange: Initial Authentication	16
2.4	Kerberos: Message Diagram	17
2.5	Kerberos: Initial Exchange of Messages	18
2.6	Kerberos: Secondary Request	19
2.7	Kerberos: Secondary Reply	20
2.8	Kerberos: Final Exchange or Messages	21
2.9	Kerberos: Authentication	22
2.10	Clock Asynchrony	24
3.1	Client	26
3.2	Authentication Server	27
3.3	Server	28
3.4	State Transition Diagram	31
3.5	Client with Oracle	39

3.6 Authentication Server with Oracle 40

3.7 Server with Oracle 41

4.1 Clock Asynchrony 43

4.2 Future Timestamp 43

4.3 Replayed Message 44

4.4 State Transition Diagram 47

4.5 State Diamond 48

4.6 Unsafe Transitions 50

4.7 Vulnerable Window 1 52

4.8 Vulnerable Window 2 53

4.9 Vulnerable Window 3 53

A.1 Random 61

CHAPTER 1

INTRODUCTION

1.1 Problem

Collections of processors networked together form a distributed system. These communicating processors may share resources and information with one another. In order to protect these resources, the processors in the system verify the identities of nodes with which they are communicating. Authentication protocols are algorithms that perform this verification. Recent increases in the interconnectivity of computers has created a need for reliable distributed authentication protocols.

The Kerberos protocol is a widely accepted and deployed distributed authentication algorithm. Kerberos depends directly on synchronized clocks. Synchronized clocks, in turn, depend on timing properties in a network. Reasoning about the Kerberos protocol, therefore, requires reasoning about both the clock and timing properties of a network. Thus, when Kerberos is deployed in networks with different synchronization properties or the synchronization properties of a network change, the correctness of this protocol must be re-analyzed. In real systems, ensuring bounds on network parameters is hard and sometimes impossible. Malicious processes may exploit vulnerabilities that are created by moments when the timing guarantees required by the Kerberos protocol are not met.

1.2 Approach

Modularity breaks designs into tractable components or modules. We can reason about each module relative to the abstract specification of other modules in the system. Modules encapsulate design decisions that are likely to change. Since Kerberos depends directly on timing parameters in the system, we cannot change the timing model without re-verifying the correctness of the authentication protocol. In order to address the vulnerabilities in the Kerberos protocol, we create a level of abstraction between the network parameters and the protocol. We can then reason about changes to the network in isolation of the protocol.

We posit an oracle that represents the current system model for Kerberos. We extend this oracle to consider a set of oracles whose specifications represent networks with different timing parameters. Using concepts from stabilization theory, we discuss the security of Kerberos with respect to these different oracles.

1.3 Thesis

A modular approach provides useful abstractions that ease reasoning and thwart attacks in authentication protocols. Oracles encapsulate timing parameters to decouple protocol correctness from timing parameters creating an effective approach for reasoning about authentication.

1.4 Organization

In Chapter 2, we present background material. In Chapter 3, we model the Kerberos protocol in Abstract Protocol notation and encapsulate the timing parameters of Kerberos with oracles. In Chapter 4, we describe the vulnerability of the protocol

and apply software engineering principals to improve reasoning about authentication. In Chapter 5, we set the context of this work among others in the field. In Chapter 6, we draw conclusions about this work.

CHAPTER 2

BACKGROUND

2.1 Authentication

Authentication is the verification of a communicating principal. A *principal* is the basic entity that participates in authentication [KNT91]. Typically, a principal represents a computer, a process, or a node in the system that is either physically or logically separated from all other principals. *One-way authentication* requires that only one principal prove its identity to another principal, while *mutual authentication* requires that both participating principals prove their identities to each other. A verifier is any principal that authenticates other principals. Authenticating a principal allows the verifier to determine what operations and information the principal can access.

The global state of a protocol encompasses the local states of the processes and the states of the channels. The set of reachable states of a protocol are those states witnessed by executing any action of the protocol. The set of safe, or legal, states are the subset of reachable states that satisfy the specification of the protocol. An *intruder* or *adversary* is an entity that performs malicious actions. When an authentication protocol is composed with an intruder, unsafe states become reachable. The actions executed by an adversary can be viewed as faults. Authentication protocols

can be classified by their fault-tolerance properties, namely, the level of tolerance a protocol achieves when composed with a specific set of adversaries.

A program can demonstrate *masking*, *nonmasking*, or *fail-safe* tolerance [AG93]. In masking fault-tolerance if a fault occurs when the protocol is in a legal state, the protocol remains in a legal state. Nonmasking fault-tolerance is less strict. If the protocol is in a legal state and a fault occurs, nonmasking tolerance allows the protocol to enter illegal states but requires the protocol eventually return to a legal state. Fail-safe fault-tolerance is again less strict than masking fault-tolerance. Fail-safe tolerance requires that a protocol always meet its safety specification, but can violate the progress specification in the presence of faults.

2.2 Abstract Protocol Notation

We summarize portions of the Abstract Protocol (AP) notation [Gou98] that will be used to model authentication protocols. The reader is directed to chapters 3 and 4 of [Gou98] for a more thorough description.

2.2.1 Processes

In this notation, a process takes the following general form:

```
process   <process name>

const     <const name>, ... , <const name>
inp       <inp name> : <inp type>, ... , <inp name> : <inp type>
var       <var name> : <var type>, ... , <var name> : <var type>

begin
          <action> [] ... [] <action>
end
```

Each *process* contains a unique name, a declaration section, and an action section.

The declaration section includes global constants, local inputs, and local variables. *Constants* can be read but not written by the process; constants with the same name in multiple processes indicate the same value. All constants are of type integer. *Inputs* and *variables* are both local to a process and, hence, similar names can take on separate values at different processes. Inputs can be read but not written, while variables can be both read and written by a process. Comments in AP delineated using curly braces. Initial conditions for a process are specified by comments in the protocol.

The action section consists of *actions* in the form of guarded commands:

guard $\rightarrow$ statement

Actions in AP can contain guards that are either local to the process, receive guards, or timeouts. We will not discuss timeout guards in this thesis. Consider a process, P, with the structure defined above. A *local guard* for P consists of a boolean expression. AP uses the '=' operator to test equality. A *receive guard* for P takes on the following form:

rcv < message> **from** < process name >

Here, process name can take on any process name in the system except P. Thus, it is impossible for P to communicate with itself via message passing.

A *message* can contain zero or more fields. The fields of a message are separated by commas. Thus, messages can take on one of the following two forms:

<message name>
<message name> (<field.0> , ... , <field.n>)

The message name is an uninterpreted symbol, and each field is interpreted. The notation $ch.P.Q\ \langle\rangle$ is used to identify the channel from P to Q. This channel consists of a sequence of message instances: $ch.P.Q\ \langle$ **msg1**; **msg2**(10); **msg3**(2,5) $\rangle$.

Where **msg1** has not fields, and **msg2** and **msg3** have one and two fields respectively. We denote the number of messages in a specific channel as $\# ch.P.Q$ and the number of a specific type of messages as $<$message name$>$ $\# ch.P.Q$.

A message instance is constructed by executing a *send* statement:

send < message> **to** <process name>

For all i, x.i is local to the process. Execution of this statement at process P creates a message instance where the values of the fields correspond to the values of x.0 through x.n and then appends this instance to ch.P.$<$process name$>$.

Statements consist of skip, assignment, sending, sequence, selection, and iteration. We have already discussed the send statement. The *skip* statement, **skip**, executes by doing nothing. The *assignment* statement denotes parallel assignment:

x.0, ... , x.n := E.0, ... , E.n

Each expression (E.0, ... , E.n) is evaluated and the value of each expression on the right hand side is assigned to the corresponding variable on the left hand side. Thus, execution of the statement results in x.0 having the value of E.0, ... , and x.n having the value of E.n.

Nondeterministic assignment can be achieved by using the expression **random**. Evaluation of the **random** expression results in a value from a uniform distribution of the domain of the corresponding variable. Thus, successive execution of x := **random** results in assigning to x each of its possible values infinitely often.

The *sequence* statement connects two statements with a semicolon:

<statement>; <statement>

This indicates that the execution of the first statement should be sequentially followed by execution of the second.

A *selection* statement has the following form, where lg.i stands for a local guard and stmt.i stands for a statement:

if lg.0 $\rightarrow$ stmt.0 [] ... [] lg.0 $\rightarrow$ stmt.0 **fi**

In a selection statement, the disjunction of the local guards must be true. Recall that local guards are boolean expressions. Execution of the selection statement begins by first evaluating all the local guards, arbitrarily selecting a true local guard, and then executing the statement corresponding to the chosen guard.

An *iteration* statement has the following form, where lg stands for a local guard and stmt stands for an a legal AP statement:

do lg $\rightarrow$ stmt **od**

Execution begins by evaluating lg.i. If the local guard is true, the statement will execute. Execution repeats until the local guard becomes false. The termination condition for the iteration statement requires that for all executions, there exists an integer, k, such that the statement is guaranteed to terminate within k iterations.

2.2.2 Security in Abstract Protocol Notation

Authentication protocols use cryptography as a means of maintaining the *integrity* of messages [Gou98]. Integrity ensures that the message received is the same as the message generated. Cryptography enables two entities to communicate on an insecure channel in such a way that no one else can understand the contents of their communications. The information to be communicated is called *plaintext*. When the plaintext is encrypted using a key, the result is called *ciphertext*. The two communicating principals share keys that allow them to encrypt the plaintext and decrypt the ciphertext so that they can understand the contents of a message. In

addition to the basic structures necessary for constructing protocols, we must consider the additional structures necessary for constructing authentication protocols. The security concepts presented here are from chapter 18 of [Gou98].

In AP, security protocols assume that each data item to be transmitted is simply an integer. Thus, plaintext is represented by data items. Cryptography is incorporated by assuming the existence of both an encryption function, NCR, and a decryption function, DCR. These functions both take a security key and data item as parameters and produce another data item. Keys, themselves, are just integers, and thus, are also data items. The data item resulting from applying the encryption function to a key, K, and data item, d, is NCR(K,d) and is described as the encryption of item d using key K. Similarly, DCR(K,d) is described as the decryption of item d using key K.

The Abstract Protocol notation defines a pair of keys (K,L) to be secure under the following two conditions:

1. *Restoration*: For every data item d,

   $$d = \mathrm{DCR}(L,\mathrm{NCR}(K,d)) \text{ and}$$

   $$d = \mathrm{DCR}(K,\mathrm{NCR}(L,d))$$

2. *Hiding*: For every key K' other than K and every key L' other than L, there is a data item d such that

   $$d \neq \mathrm{DCR}(L',\mathrm{NCR}(K,d)) \text{ and}$$

   $$d \neq \mathrm{DCR}(K',\mathrm{NCR}(L,d))$$

Restoration requires that data item d encrypted with key K and decrypted with key L result in the original data item. This requirement indicates that the plaintext

created by decryption at the receiving principal is indeed the plaintext generated by the sending principal. Hiding requires that the only principals able to see the plaintext message will be those holding the secure keys. A secure key pair is *symmetric* (or shared) if and only if K = L. The protocol we consider uses symmetric keys.

Nonces are used in authentication protocols to identify messages or create a sequential view on the order of messages. In AP notation, a nonce is represented as an integer or data item. Each process can generate nonces by using a function, NNC. The NNC function at each process creates a sequence of nonces satisfying:

1. Unpredictability: The value of any generated nonce cannot be deduced from the values of previously generated nonces.

2. Nonrepetition: Each generated nonce has a unique value.

In security protocols, two types of information appear in channels: ciphertext and plaintext. We represent ciphertext in a channel by denoting the key used to encrypt the message. For shared keys, we use $K_{P,Q}\langle\rangle$ where the names of the processes that share the key are listed in curly braces and the information inside angled brackets. Hence, $K_{P,Q}\langle i\rangle$, indicates a piece of information, i , encrypted with a key shared by P and Q .

2.3 Kerberos

> $\kappa\varepsilon\rho\beta\varepsilon\rho o\sigma$; also spelled Cerberus. "n. The watch dog of Hades, whose duty it was to guard the entrance–against whom or what does not clearly appear; . . . is known to have had three heads . . . " –Ambrose Bierce, *The Enlarged Devil's Dictionary* [MNSS87]

The Kerberos authentication service [KNT91] was developed at the Massachusetts Institute of Technology to protect the network services of Project Athena [CDEGR90].

The first three versions of Kerberos were created and used internally. Versions 4 and 5 are currently in use by many systems [NT94]. In this thesis we focus on version 5 of the protocol and use the terms Kerberos and Kerberos protocol to indicate version 5.

Kerberos contains three non-malicious entities: a *Client* (C), an *Authentication Server* (AS), and a *Server* (S). The goal of Kerberos is for S to authenticate C. Implicit in this goal is a subtle requirement, namely that S should not authenticate any other entity as C. Each entity trusts the Authentication Server to generate original, secure keys. Thus, the Authentication Server is a trusted source and shares a key with each principal in the system. When describing the Kerberos protocol, we will use K to indicate a Key and subscripts to show ownership of a key. Thus, $K_{C,AS}$ indicates a key shared by C and AS. The Client uses a *ticket* and *authenticator* to prove its identity to the Server and to establish a temporary communication key that can be used to create a secure channel between the two principals. A ticket consists of a freshly generated key, the name of the requesting principal, the time of creation, and a lifetime for which the ticket is valid. An authenticator contains an encrypted timestamp.

The figures below illustrate the Kerberos protocol [Tun99]. The principals participating in each step of the protocol are pictured on each line. The arrow represents information passing from one principal to another. Thus, when the arrow is present both a sending and receiving principal are pictured. A single principal on a line indicates local processing at that node. Diamonds are used to represent the local decisions being made at a process. Boxes are used to represent the sets of information being sent between principals. The boxes are labeled for easy identification.

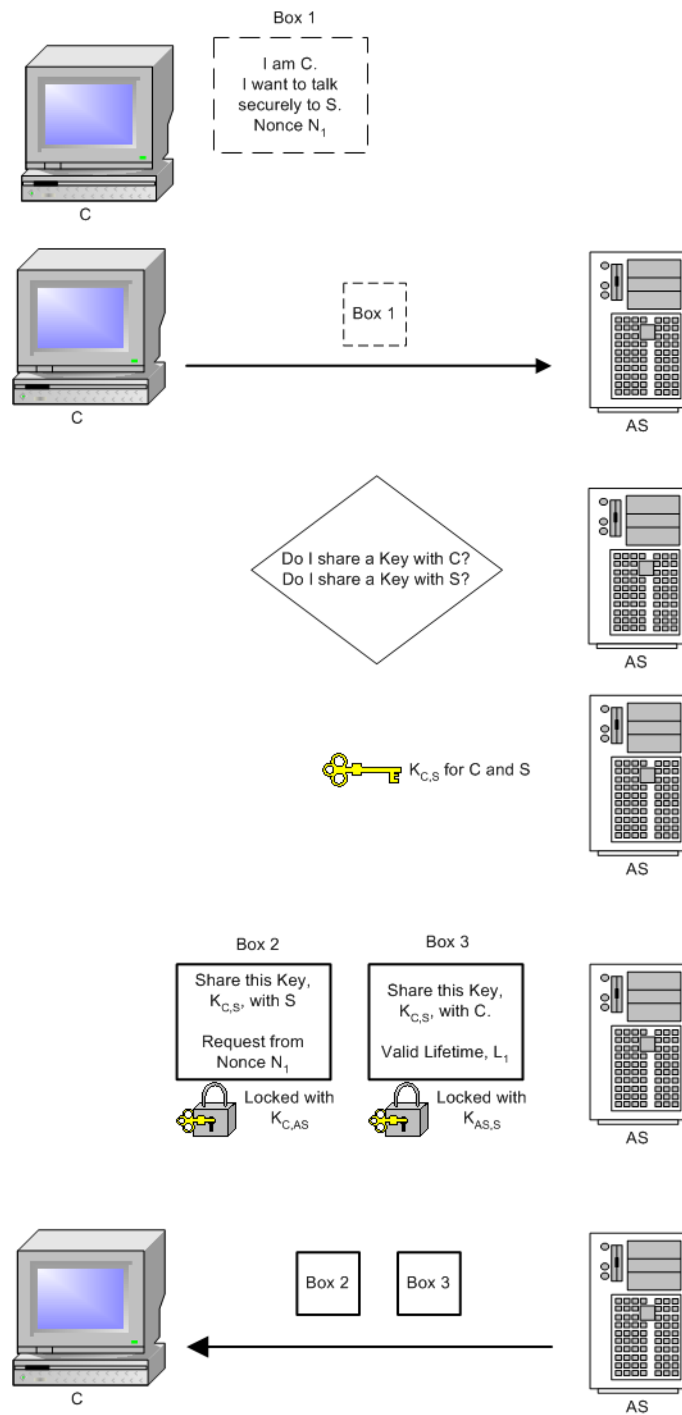

Figure 2.1: Simple Exchange: Initial Exchange of Messages

Plaintext is sent using an unlocked box (a dashed line). Ciphertext is pictured as information inside a box with a lock (a solid line). The lock can only be opened by a principal that holds the key used to lock the box. Thus, information in unlocked boxes can be read by all principals and information in locked boxes can be read only by principals possessing the specified key.

Kerberos adds additional protection by breaking the Authentication Server into logically distinct sectors, the *Key Server* (KS) and *Ticket Server* (TS). Since both KS and TS are part of AS, they are trusted by all other entities. Tickets are generated for use in the protocol by both of these sectors. We first consider a protocol that requires C to obtain only one ticket, from AS, to prove its identity to S. We then demonstrate an additional layer of protection that requires C to receive tickets from each sector, before C can prove its identity to S.

2.3.1 Simple Exchange Protocol

The initial exchange of messages is pictured in Figure 2.1. In the first message, C requests a ticket from AS. This message is represented by Box 1. Box 1 contains unencrypted information that can be seen by all principals. Box 1 includes the name of the node generating the message, C, the name of the node with which C wants to communicate, S, and a nonce, N_1. Upon receipt of Box 1, AS verifies that it shares a key with both C and S. Recall that initially each principal in the system shares a key with AS and that $K_{C,AS}$ is the key shared between C and AS. AS will only generate tickets for those principals in the system with which it shares a key.

Once this check completes successfully, AS creates the requested ticket. In the second step of the protocol, AS generates a fresh key, $K_{C,S}$, for C and S to share. The second message contains Boxes 2 and 3, constructed from the information in Box 1 and the generated key, $K_{C,S}$. AS creates Boxes 2 and 3 in response to the original request from C. Box 2 provides C with the session key $K_{C,S}$, enclosing the key and the nonce, taken directly from Box 1. This indicates to C that Box 1 was received and Boxes 2 and 3 were created in response.

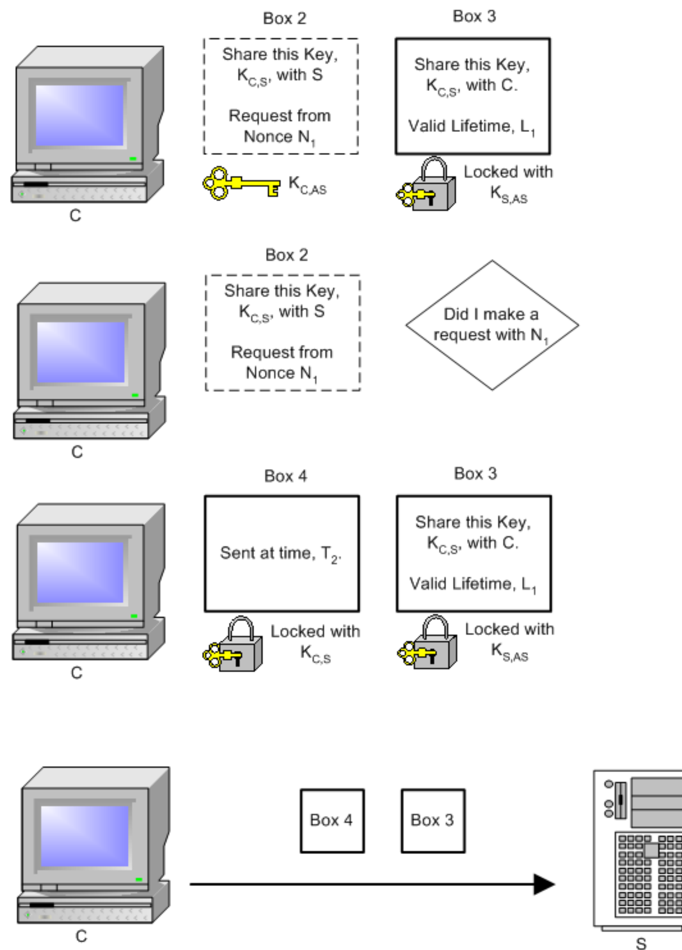

Figure 2.2: Simple Exchange: Final Exchange of Messages

Box 3 is a ticket to be forwarded to S. The ticket includes the freshly generated key, $K_{C,S}$, the name of the requesting principal, C, and a lifetime for which the ticket is valid, L_1. Notice that the ticket, Box 3, is locked with $K_{AS,S}$. Hence, C cannot see the information inside the ticket, but it is needed in order for C to authenticate with S. In the Kerberos model, each ticket contains a finite time interval called a *lifetime*, L. Each ticket is, therefore, only valid until the given lifetime expires; future authentications will require the generation of a new ticket. Boxes 2 and 3 comprise Message 2 and are sent back to C. The last step in Figure 2.1 shows the sending of Boxes 2 and 3 from AS to C.

Figure 2.2 illustrates the creation of Message 3 at C. Upon receipt of Boxes 2 and 3, C can unlock Box 2 and retrieve the information inside. The first step in Figure 2.2 shows that C can view the information in Box 2, but cannot unlock Box 3. Once C views Box 2, it verifies that it sent the original message using nonce N_1. C then uses an *authenticator*, Box 4, to prove that it is in possession of the temporary key, $K_{C,S}$, that was generated by AS. The authenticator is simply a timestamp encrypted with the temporary key.

Finally, Figure 2.3 shows the receipt of Message 3 at S. S holds $K_{AS,S}$ and can unlock Box 3, the ticket. S tests the ticket in Box 3 to determine its validity. If the ticket is valid, S accepts the session key in Box 3 and unlocks Box 4. S tests the timestamp in Box 4 for recent generation, comparing the current time at S to the timestamp. If the timestamp was within an acceptable bound, S *authenticates* C. If the timestamp was not acceptable, C's request is denied.

2.3.2 Additional Protection

Kerberos creates an extra layer of protection by dividing AS into two distinct entities: a *Key Server* (KS) and a *Ticket Server* (TS). KS replaces AS in the initial exchange and TS replaces S. Now KS shares secret keys with TS and C, but not with the Server. Conversely, TS shares a secret key with KS and S, but not with C. KS and TS are logically distinct services, but often reside in the same physical location. The additional abstraction creates a separation of

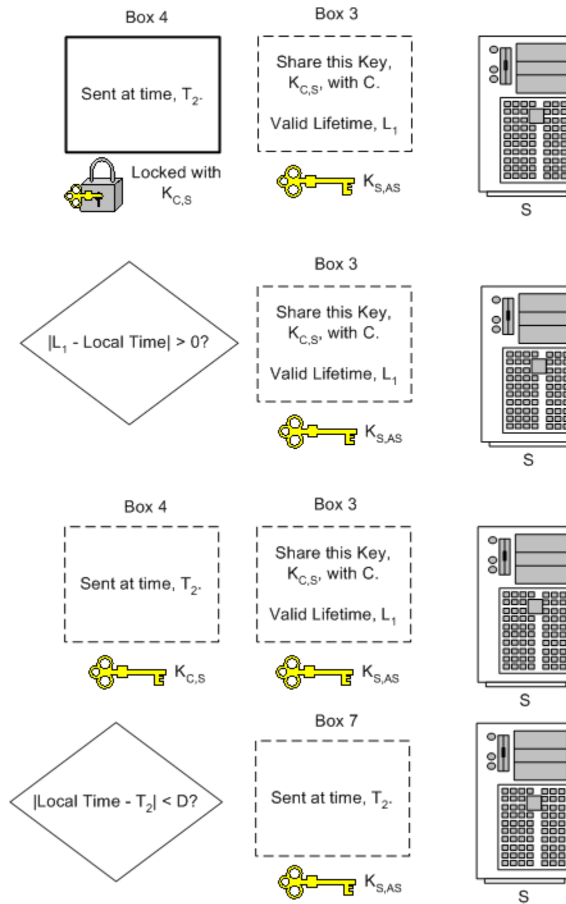

Figure 2.3: Simple Exchange: Initial Authentication

keys shared with clients and servers. That is, initially, no single entity shares a key with both clients and servers. The Kerberos protocol contains 5 messages; figure 2.4 shows the message flow among the four entities.

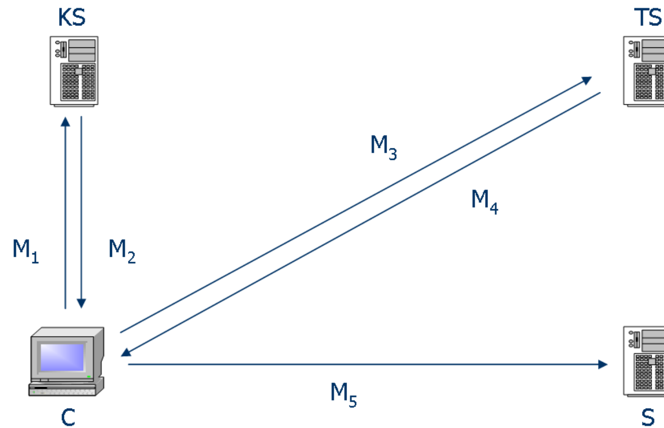

Figure 2.4: Kerberos: Message Diagram

In Figure 2.5 we see an interaction similar to that described previously. C makes an initial request for communication with TS instead of S. KS checks for shared keys with C and TS. If the check passes, a fresh key is generated, $K_{C,TS}$, for C and TS to share. Again Boxes 2 and 3 are generated from the information in Box 1 and the new key $K_{C,TS}$. This time the boxes are locked with $K_{C,KS}$ and $K_{KS,TS}$ respectively. Box 2 can be unlocked by C because C shares key $K_{C,KS}$ with KS.

C makes the similar checks and again creates an authenticator, Box 4. Appended to Message 3 is a new box, Box 5. This indicates to TS that C would like to establish a secure channel of communication with S. Message 3 includes a nonce

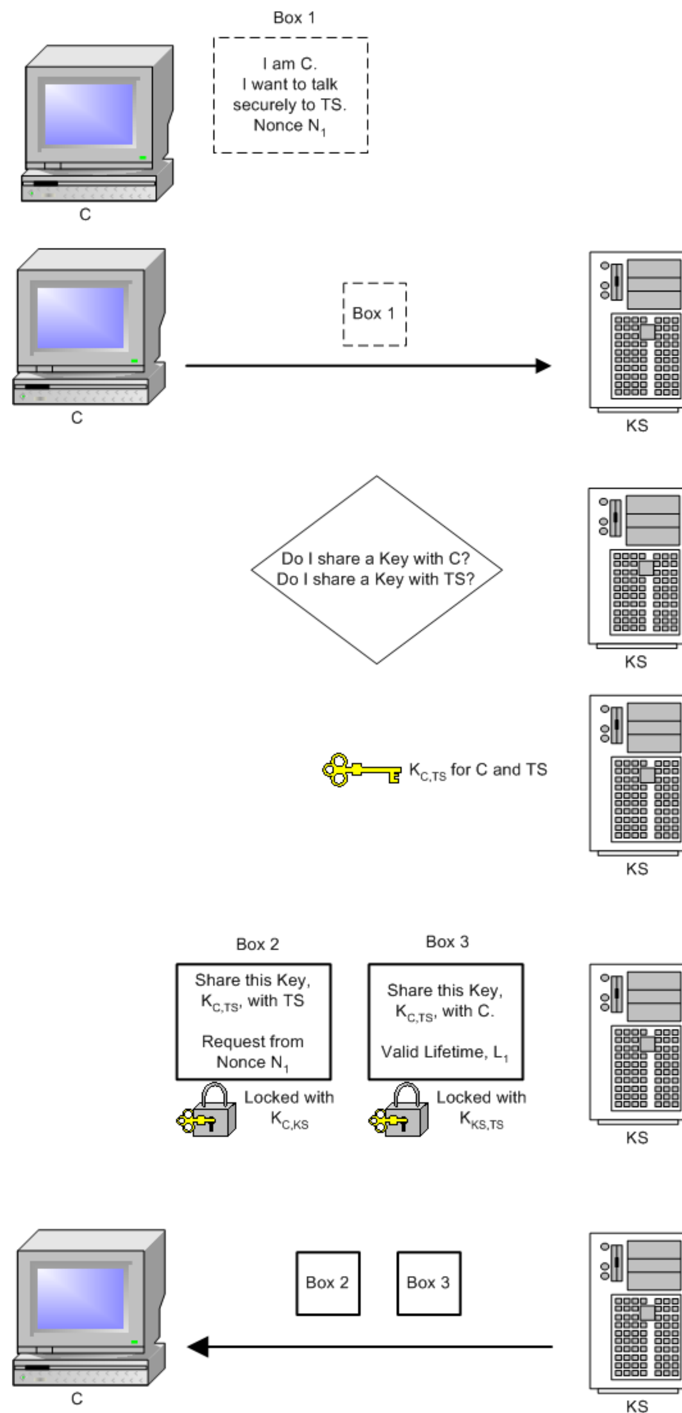

Figure 2.5: Kerberos: Initial Exchange of Messages

to be returned in Message 4 by TS. Additionally, TS takes the place of S and Message 3 is now directed to TS, see Figure 2.6.

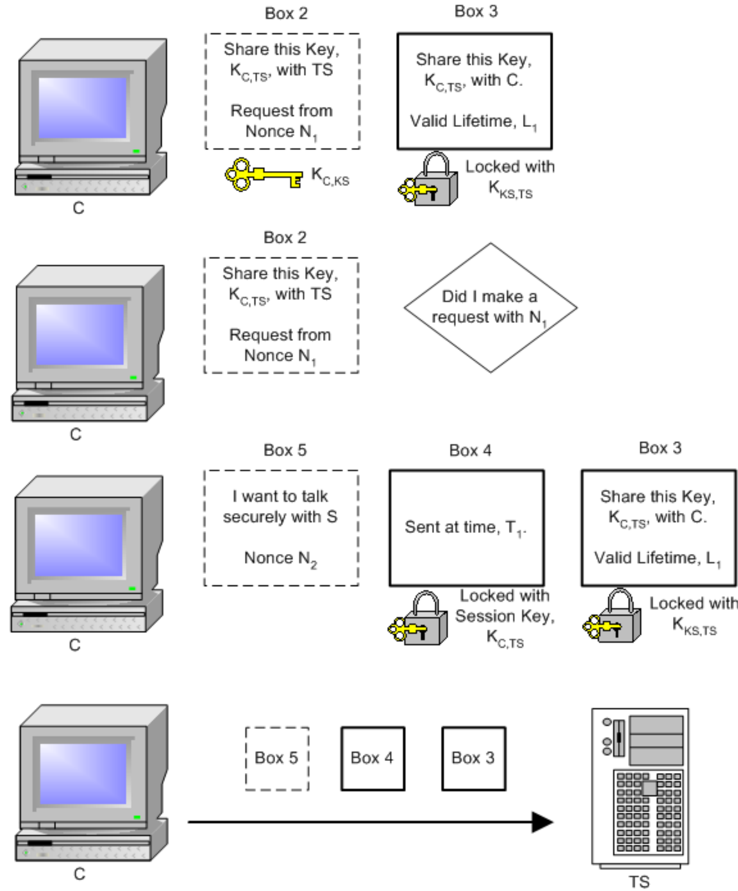

Figure 2.6: Kerberos: Secondary Request

The protocol continues in Figure 2.7, as TS verifies the ticket, authenticator, and session request from Message 3. Similar to the simple exchange protocol, TS checks that the ticket's lifetime, L_1, is still valid, that the authenticator, Box 4, was freshly generated, and that it holds $K_{C,TS}$ and $K_{TS,S}$. If all checks pass, TS generates a fresh key for C and S to share. The key, $K_{C,S}$, intended for C is placed in

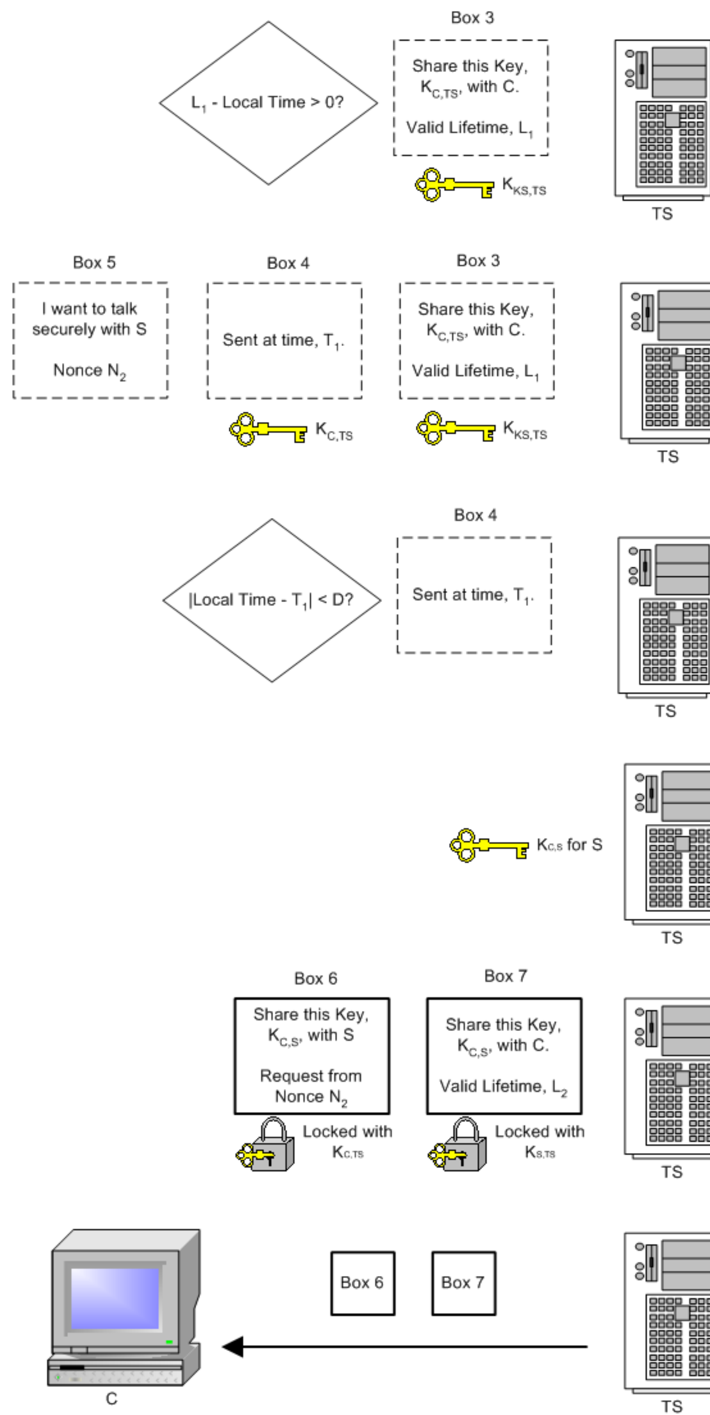

Figure 2.7: Kerberos: Secondary Reply

Box 6 along with the nonce from Message 3. A new ticket is created for S, Box 7, including the key, $K_{C,S}$, name of requester, C, and lifetime, L_2. Message 4, containing Boxes 6 and 7 are then sent back to C.

After receiving Message 4 (Figure 2.8), C tries to establish a communication session with S, by sending the fifth and final message. C unlocks Box 6 using the key it shares with TS, $K_{C,TS}$. C checks the contents of Box 6 to determine if this box was created in response to a message by verifying the nonce, N_2. Once the

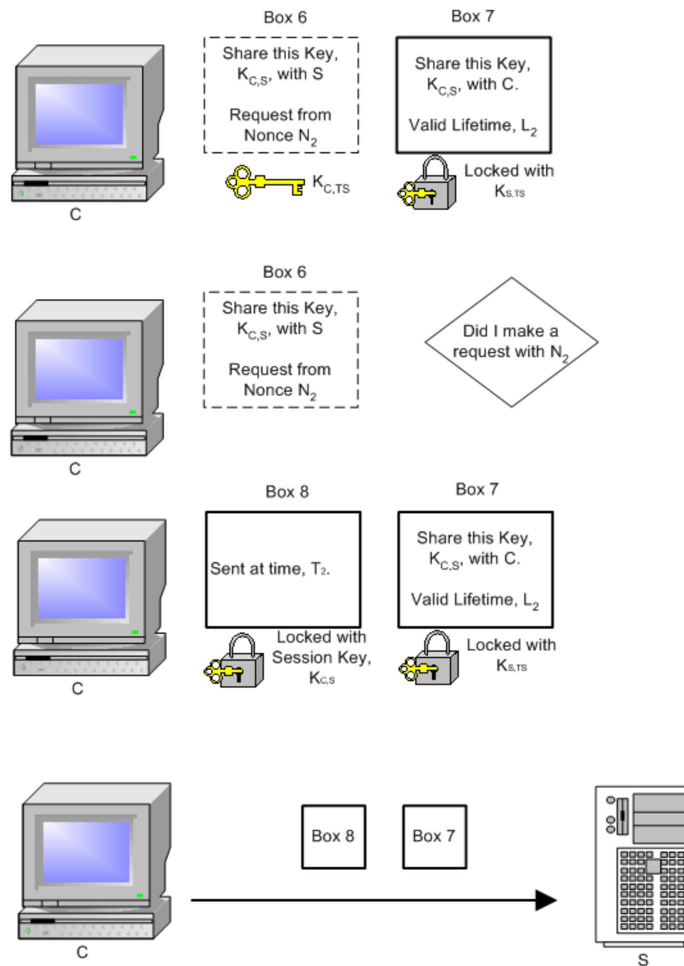

Figure 2.8: Kerberos: Final Exchange or Messages

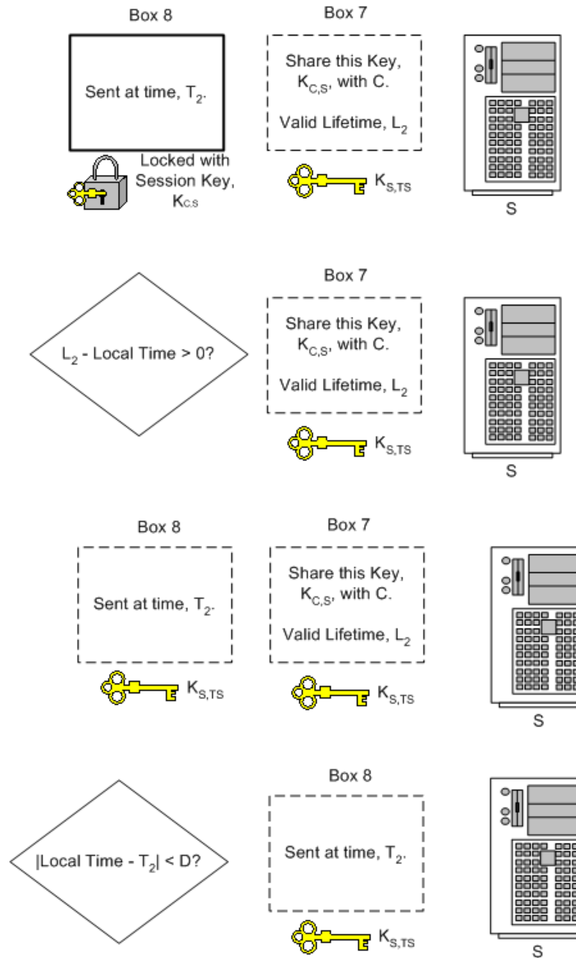

Figure 2.9: Kerberos: Authentication

nonce has been verified, C creates a new authenticator to prove to S that C holds the session key inside the ticket, Box 7. The new authenticator contains the most recent time at C, T_4. Message 5 consists of the new authenticator, Box 8, and the ticket from TS, Box 7, and is sent to S.

Figure 2.9 displays Message 5 arriving at S, and S unlocking the ticket, Box 7, using the key shared with TS, $K_{TS,S}$. S inspects the information in the ticket to determine its validity. If the ticket is found to be valid, S accepts $K_{C,S}$ and

unlocks Box 8. The timestamp, T_4, in Box 8 is checked for recent generation. If the timestamp is within the required window, S authenticates C.

This description presents a successful run of the protocol. If any of the checks along the way do not pass, the protocol terminates at that point. If the protocol terminates, then C must restart the protocol by initiating a new request to the KS. More detailed discussions of Kerberos can be found in [SNS88], [MNSS87], [Neu], and [KPS02].

2.4 Suppress-Replay

Kerberos depends on the synchronization of clocks between all nodes in the system. Synchronized clocks allow principals to check the timestamps generated by other principals and have an expectation of the range in which the timestamp should occur. Kerberos has often been criticized for its claimed reliance on "loosely synchronized" clocks [MNSS87] [Neu]. The imprecise definition does not discuss the security or implementability issues for Kerberos. This synchronization requirement creates restrictions on the kinds of networks in which Kerberos can safely be deployed.

We model moments when clocks are not synchronous as a *fault* in the system. A fault occurs when the clocks at two processes drift apart by more than some acceptable difference, δ. Owing to the vague description of requirements in the Kerberos protocol, asynchronous clocks may be a fault in the system or may be a flaw in the design. The following figure models the occurrence of a fault at node C and the correction of the fault. In the timing diagram, the x-axis represents real time. The y-axis represents the difference between the local time of a process and real time. Thus, when the clock at a process is synchronized with real time, the

graph of that process is a horizontal line. When the clock at a process advances faster than real time, the graph has a positive slope. Conversely, when a clock at a process advances slower than real time, the graph has a negative slope. If local clocks are monotonic, the slope of the graph is always greater than, or equal to, -1. Figure 2.10 shows the clocks of S and C. S remains synchronized with real time, while C moves ahead of real time, hits a plateau, and then re-synchronizes with real time. These diagrams will be used in the thesis to discuss the actions of Kerberos with respect to time.

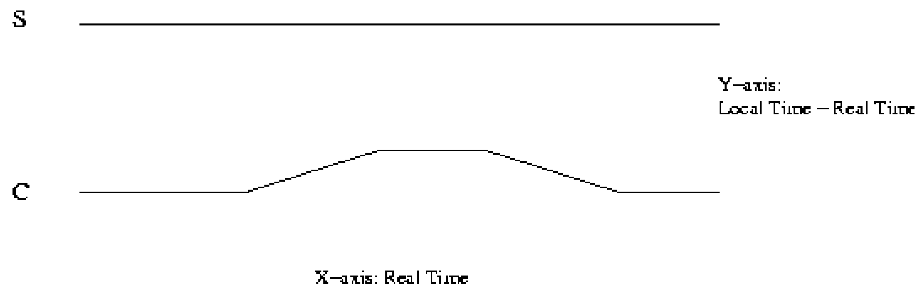

Figure 2.10: Clock Asynchrony

CHAPTER 3

MODELING KERBEROS

3.1 Kerberos in Abstract Protocol (AP) Notation

Our realization of Kerberos in AP notation consists of three processes, C, AS, and S. AS will have distinct actions representing KS and TS. We introduce the type PName for process names in the Kerberos protocol; a PName is an integer. Thus, C, KS, TS, S, and AS are all PNames. Each process contains material variables, as well as, catalytic variables. The material variables are imperative to the functionality of Kerberos. The catalytic variables provide temporary storage for a process; essentially we do not care what value these variables have at any state of the protocol. As such, only the material variables are considered in the state predicates. S contains a local variable, $auths$, representing authentication of C. If at the end of a session of the Kerberos protocol $auths = true$, then S received a valid **rqst** message sent from C in that session. However, if at the end of a session $auths = false$, no conclusion can be drawn. Thus, Kerberos must satisfy the following safety property:

$$auths \rightarrow S \text{ received a valid message sent from } C \text{ in that session}$$

```
process C

const
    KS, TS
var
    startc   :  boolean,  {initially, startc = false}
    keyc     :  array[PName] of integer,
                 {initially, keyc[KS] = K_{C,KS} ∧
                  keyc[TS] = ⊥ ∧ keyc[S] = ⊥}
    nc       :  integer,   {initially, nc = ⊥}
    x,y,z    :  integer

begin
~startc →
    nc, startc := NNC, true;
    send krqst(C, TS, nc) to AS
[]
rcv krply(x, y) from AS →
    (x, z) := DCR(keyc[KS], x);
    if z ≠ nc →  startc, nc := false, ⊥
    [] z = nc →
        keyc[TS], nc := x, NNC;
        send trqst(NCR(keyc[TS], TIM), y, S, nc) to AS
    fi
[]
rcv trply(x, y) from AS →
    (x, z) := DCR(keyc[TS], x);
    if z ≠ nc → startc, nc, keyc[TS] := false, ⊥, ⊥
    [] z = nc →
        keyc[S], nc := x, ⊥;
        send rqst(NCR(keyc[S], TIM), y) to S
    fi
end
```

Figure 3.1: Client

In our realization, we use the ⊥ symbol to represent the value of an undefined variable. In AP, this symbol might be realized as a specific integer value that is

```
process AS
const
    KS, TS
inp
    keyks          :  array[PName] of Keys,
                       {keyks[TS] = K_{KS,TS} ∧
                        keyks[C] = K_{C,KS} ∧ keyts[S] = ⊥}
    keyts          :  array[PName] of Keys,
                       {keyts[KS] = K_{KS,TS} ∧
                        keyts[C] = ⊥ ∧ keyts[S] = K_{TS,S}}
    L              :  integer,
    D              :  integer
var
    keyas          :  integer,  {initially, keyas = ⊥}
    A              :  integer,
    u,v,w,x,y,z    :  integer

begin
rcv krqst(x, y, z) →
    if keyks[x] = ⊥ ∨ keyks[y] = ⊥ → skip
    [] keyks[x] ≠ ⊥ ∧ keyks[y] ≠ ⊥ →
        keyas := Random;
        send krply(NCR(keyks[x], (keyas, z)),
           NCR(keyks[y], (x, TIM + L, keyas))) to x
        keyas := ⊥
    fi
[] rcv trqst(w, x, y, z) →
    (u, v, x) := DCR(x, keyts[K]);
    if ((v − TIM) <= 0) → skip
    [] ((v − TIM) > 0) →
        A := DCR(w, x);
        if (ABS(TIM − A) >= D) → skip
        [] (ABS(TIM − A) < D)) →
            keyas := Random;
            send trply(NCR(x,(keyas, z)),
              NCR(keyts[y], (u, TIM + L, keyas))) to u
            keyas := ⊥
        fi
    fi
end
```

Figure 3.2: Authentication Server

unattainable using the **random** expression. Since **random** can return any value selected from the domain of the variable, we use the **Random** function to indicate a function **random** that will return values selected from the domain - ⊥. Please see Appendix A for further discussion on this topic and the realization for **Random**.

```
process S
const
    KS, TS
inp
    D           :  integer
var
    auths       :  boolean,  {initially, auths = false}
    keys        :  array[PName] of integer,
                    {initially, keys[KS] = ⊥ ∧
                     keys[TS] = K_{TS,S} ∧ keys[C] = ⊥}
    A           :  integer,
    v,w,x,y,z   :  integer

begin
rcv rqst(v, w) ->
    w, x, y := DCR(keys[TS], w);
    if ((x - TIM) <= 0) -> skip
    [] ((x - TIM) > 0) ->
        A := DCR(y, v);
        if (ABS(TIM - A) >= D ∨  keys[C] ≠ ⊥) →
            skip
        [] (ABS(TIM - A) < D ∧ keys[C] = ⊥) →
            keys[C], auths  := y, true;
        fi
    fi
end
```

Figure 3.3: Server

In order to model the Kerberos protocol in AP notation, we introduce a new function TIM. TIM returns the value of the clock at a process. Different instantiations of TIM would result in different clock models. Instantiating TIM with the **random** function produces a clock whose values are taken at random from the set of integers. Instantiating TIM with sequence numbers provides a monotonically increasing clock model.

Kerberos uses time to uniquely identify messages, through timestamps, as well as to check timestamps and lifetimes for validity. Kerberos requires that each process have access to time and that the times at processes should be "loosely synchronized" [MNSS87],[Neu]. This synchronization creates a predictable window for timestamps received in messages. In the Kerberos system, without intruders, the synchronization of clocks does not affect the security of the protocol. It does, however, affect the progress guaranteed by the protocol. Kerberos is guaranteed to make progress if the summation of the time bound between the clocks of two processes, δ_1, and the message delay, δ_2, is within an acceptable bound, D. That is, progress is guaranteed only if: $\delta_1 + \delta_2 < D$. In addition, progress depends on the ticket lifetime, L, satisfying: $\delta_1 + 2\delta_2 < L$. Owing to this direct dependency on time, the progress properties for Kerberos must include these bounds explicitly. We refer to the value of the clock of a process, P, at real time x as: $time_{P,x}$. If no x is present, then $time_P$ refers to the current time at process P. Kerberos satisfies the following progress property:

$$(\#ch.C.S > 0 \ \wedge \ \forall x : |time_{C,x} - time_{S,x}| < \delta_1 \ \wedge$$
$$\delta_1 + \delta_2 < D \ \wedge \ \delta_1 + 2\delta_2 < L) \mapsto auths$$

If a state is reached where the number of messages in $ch.C.S$ is greater than zero, the clocks at C and S are synchronized within some bound, δ_1, and message delay is within δ_2, then, eventually ($\mapsto$) $auths$ will be set to true. In short, if the system obeys the synchronization requirements of Kerberos and C generates the last message of the protocol (C is the only process capable of generating this message), S will authenticate C.

3.1.1 States of Kerberos

The AP representation of Kerberos is shown in Figures 3.1, 3.2, and 3.3. We discuss the material variables in the representation, the states generated by the actions at each process, and the corresponding state diagram.

Process C has 3 material variables:

$$startc \equiv \text{authentication procedure has started}$$

$$keyc \equiv \text{keys held by } C \text{ initially or received in messages}$$

$$nc \equiv \text{nonce sent in the current exchange}$$

Process S has 2 material variables:

$$auths \equiv \text{authentication was successful between } C \text{ and } S$$

$$keys \equiv \text{keys held by } S \text{ initially or received in messages}$$

Process AS has no material variables. Each of the five messages of Kerberos have distinct message types, **krqst**, **krply**, **trqst**, **trply**, **rqst**. These message types have a one-to-one correspondence to the informal message in Kerberos as follows:

krqst	Message 1 (M_1)
krply	Message 2 (M_2)
trqst	Message 3 (M_3)
trply	Message 4 (M_4)
rqst	Message 5 (M_5)

These message types capture the request-reply handshake between C and KS (**krqst**, **krply**), the request-reply handshake between C and TS (**trqst**, **trply**), and C's request to S (**rqst**).

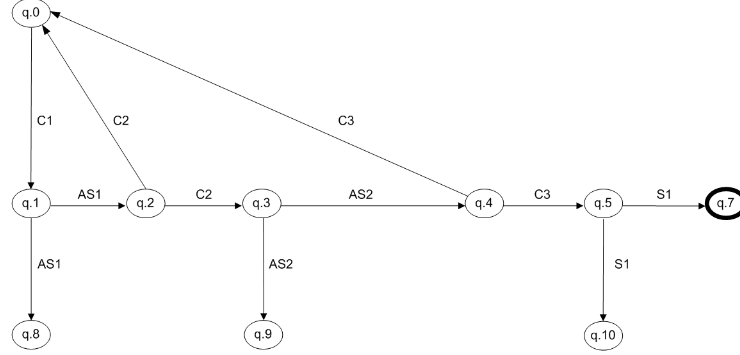

Figure 3.4: State Transition Diagram

We identify states **q.0-q.10** as the reachable states for the Kerberos protocol. The transitions among these states are shown in Figure 3.4. Each transition contains a letter and a number. The letter indicates the process performing the action; the number indicates which action at the process is being executed. All steps, except the first, are triggered by the receipt of a message. As such, initially the only enabled step is the creation of Message 1 in C. **q.0** shows the initial state of the Kerberos system. Since all actions except action 1 of C are guarded by the receipt of messages and initially all channels are empty, none of these actions are enabled. Action 1 of C is guarded by a boolean expression that is initially $true$.

q.0: $ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$

$$keyc[TS] = keyc[S] = keys[C] = keys[KS] \wedge nc = \perp$$

$$keyc[KS] = K_{C,KS} \wedge keys[TS] = K_{TS,S} \wedge \sim startc \wedge \sim auths$$

We note here that some variables do not change by executing any actions. We use Z to represent those variables in the following states.

$$Z \equiv keyc[KS] = K_{C,KS} \wedge$$
$$keys[KS] = \perp \wedge keys[TS] = K_{TS,S}$$

Thus, after substitution:

$$\textbf{q.0:}\ ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$\sim startc \wedge nc = \perp \wedge \sim auths \wedge Z$$

Executing action 1 of C , the only enabled action, results in updating the *startc* variable to indicate that the protocol has begun, as well as generating a nonce for the current round of communication. Next, C appends message 1 of the protocol to channel $ch.C.AS$. Since this is the first action to execute, no prior messages exist in $ch.C.AS$. Thus, action 1 of C creates message 1 and sends it to AS. Executing action 1 of C transitions from **q.0** to **q.1**, step 2 in Figure 2.5.

$$\textbf{q.1:}\ ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\ \rangle \wedge$$
$$ch.C.AS = \langle krqst(C,\ TS,\ N_1)\rangle \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge nc = N_1 \wedge \sim auths \wedge Z$$

From **q.1** we see that $\sim startc$ is false (disabling action 1 of C) and incoming channels to C and S are empty (disabling actions at those processes). $ch.C.AS$, however, contains a single message of type **krqst**, enabling action 1 of AS. Action

1 of AS accomplishes steps 3-6 in Figure 2.5. Upon receiving the **krqst** message, AS checks to determine if it shares keys with the process names in fields 2 and 3 of the **krqst** message. If it does not, a **skip** is executed and the protocol will terminate in state **q.8**. In **q.8**, no further actions are enabled (all channels are empty and $startc$ is false) and the protocol terminates.

$$\textbf{q.8:}\ ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge nc = N_1 \wedge \sim auths \wedge Z$$

In a successful run of action 1 of AS, this check passes. AS will then generate a new key, and compose M_2, a message of type **krply**, using the new key and the information from M_1. This message is then sent to C and is in transition in state **q.2**.

$$\textbf{q.2:}\ ch.C.AS = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\ \rangle \wedge$$
$$ch.AS.C = \langle krply(K_{C,KS}\{K_{C,TS}, N_1\},\ K_{KS,TS}\{C, L_1, K_{C,TS}\})\rangle \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge nc = N_1 \wedge keyas = K_{C,TS} \wedge \sim auths \wedge Z$$

Again from **q.2**, only action 2 of C is enabled. C decrypts the first field of the **krply** message; C does not hold a key to decrypt the second field. C checks the **krply** message for the nonce included in the original request. If the nonces do not match, the protocol returns to the initial state, **q.0**. If the nonces match, C accepts the key, creates a fresh the nonce value, generates a **trqst** message, and sends the

message to AS. The new state is **q.3**.

$$\textbf{q.3:}\ ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\ \rangle \wedge$$

$$ch.C.AS = \langle trqst(K_{C,TS}\{T_1\},\ K_{KS,TS}\{C, L_1, K_{C,TS})\}, S, N_2)\rangle \wedge$$

$$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$$

$$startc \wedge nc = N_2 \wedge \sim auths \wedge Z$$

The **trqst** message in $ch.C.AS$ includes an authenticator, a ticket, a process name, and a nonce, and it enables action 2 of AS. AS receives the message, first decrypting the ticket (field 2). If the lifetime of the ticket has expired, a **skip** executes and the protocol will terminate in state **q.9**. If the ticket is valid, AS uses the key inside the ticket to decrypt the authenticator (field 1). The timestamp, or unencrypted authenticator, is checked for recent generation (within D time units). If the timestamp was not generated within D units, a **skip** executes and the protocol will terminate in state **q.9**. If both the lifetime and timestamp checks are successful, the protocol transitions into **q.4**, appending a **trply** message to the channel between AS and C.

$$\textbf{q.9:}\ ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$$

$$keyc[TS] = K_{C,TS} \wedge keyc[KS] = keys[C] = \perp \wedge$$

$$startc \wedge nc = N_2 \wedge \sim auths \wedge Z$$

$$\textbf{q.4:}\ ch.C.AS = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\ \rangle \wedge$$

$$ch.AS.C = \langle trply(K_{C,TS}\{K_{C,S},\ N_2\},\ K_{TS,S}\{C,\ L_2,\ K_{C,S}\})\rangle \wedge$$

$$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$$

$$startc \wedge nc = N_2 \wedge \sim auths \wedge Z$$

From **q.9**, no actions are enabled and the protocol terminates. **q.4** enables a single action, namely action 3 of C. This action is almost identical to action 2 of C. C receives the **trply** message and can, again, only decrypt the first portion. After checking the nonce, C resets $startc$, nc, and $keyc[TS]$ if the nonce is not valid and the protocol returns to state **q.0**. Otherwise, C stores the key from the **trply** message and resets the nonce. C generates the final message using the ticket from the **trply** message and creating a fresh timestamp encrypted with the key from message **trply**. This message is then sent to S by placing it in the channel shared by C and S (**q.5**).

$$\textbf{q.5:}\ ch.C.AS = ch.AS.C = ch.S.C = ch.AS.S = ch.S.AS = \langle\ \rangle \wedge$$
$$ch.C.S = \langle rqst(K_{C,S}\{T_2\},\ K_{TS,S}\{C,\ L_2,\ K_{C,S}\})\rangle \wedge$$
$$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \bot \wedge$$
$$startc \wedge nc = \bot \wedge \sim auths \wedge Z$$

From **q.5**, action 1 of S is the only enabled action. S receives the **rqst** message from C and decrypts the ticket (field 2). After decrypting, it tests the lifetime (field 2 of the ticket) for validity. If the ticket has expired, the protocol terminates in **q.10** without authentication.

$$\textbf{q.10:}\ ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$$
$$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \bot \wedge$$
$$startc \wedge nc = \bot \wedge \sim auths \wedge Z$$

If the ticket is valid, S uses the key inside the ticket to decrypt the authenticator (field 1 of **rqst**). The timestamp in the authenticator is checked for generation

within D time units. If the timestamp is not valid, again the protocol terminates in **q.10** without authentication. If the timestamp is valid, S accepts the key for communication with C and sets $auths$. A successful execution of this action will result in authentication of C and reaches **q.7**. Notice that we do not include a state labeled **q.6**. Later, we discuss splitting state **q.5** into two separates states.

$$\textbf{q.7:}\ ch.C.AS = ch.AS.C = ch.C.S = ch.S.C = ch.AS.S = ch.S.AS = \langle\rangle \wedge$$
$$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = K_{C,S} \wedge$$
$$startc \wedge nc = \perp \wedge auths \wedge Z$$

3.1.2 Correctness

Proof of Safety. Initially, $auths$ is false. Action 1 of S is the only action that writes $auths$. Thus, all other actions preserve safety trivially by not updating the value of $auths$. C is the only principal who creates messages of type **rqst**. Hence, when a message of type **rqst** is received it must have been sent from C. Upon receipt of a **rqst** message, S tests the validity of the message by first checking the lifetime on the ticket and then checking the timestamp in the message. If both these checks pass, then S has received a valid message from C and S sets the value of $auths$ to true. Action 1 of S is the only action that sets $auths$ to true and it does so only upon receipt of a valid **rqst** message sent from C. No action changes $auths$ to false.

Proof of Progress. Assume the Kerberos system meets the synchronization requirement, *i.e.* $\forall x : |time_{C,x} - time_{S,x}| < \delta_1 \wedge \delta_1 + \delta_2 < D \wedge \delta_1 + 2\delta_2 < L$. Then in state **q.5**, C places a valid message into the channel. This message will be received at S in less than δ_2 time units, meaning action 1 of S must execute

within the allotted time. The synchronization requirements guarantee that both checks pass and S authenticates C. Thus, if state **q.5** is reached by the protocol, eventually, S authenticates C.

3.2 Oracle

In order for the Kerberos protocol to make progress, a bound must be known on both relative clock values and message delay. Implicit in this conclusion are assumptions about the way time is kept and the timing parameters in the network. The progress of Kerberos relies not only on the existence of these bounds, but that the bounds be known to the processes. These strict requirements imply that Kerberos can only make progress in a *synchronous system* with *k-synchronized* clocks. A synchronous system is one with known bounds on message transmission and relative processor speed. Clocks are k-synchronized if the values of two clocks always remain within k time units of one another. The Kerberos designers provide no intuition about the guarantees made in a system with less stringent timing properties. In addition, when we introduce intruders into the system, these restrictions create vulnerabilities not only for progress, but also for safety.

We would like to reason about Kerberos independent of the specific timing parameters in a system. By encapsulating these parameters in an oracle, we can reason about Kerberos based on an abstract representation of the timing parameters in the system. We posit an oracle that provides this abstraction. By hiding the timing information in the system, we are able to discuss the correctness properties of Kerberos relative the abstract specifications of an oracle rather than discussing specific models of time.

We create an oracle that hides the model of time being used in the above protocol. The oracle uses a notion of suspicion. An oracle will suspect a process P if the timestamp T is outside a bound B. The oracle takes three parameters, the first two are taken from a message and the third is a property of the protocol.

P - process name included in the message
T - timestamp included in the message
B - acceptable upper bound on timestamp

We specify the oracle for a process Q as follows:

$$\textbf{Suspect(PName P, Timestamp T, Bound B)} \Leftrightarrow |time_Q - T| > B$$

If a process, Q, suspects a process, P, then the time at Q differs from the timestamp created by P by more than bound B, $|time_Q - T| > B$. Additionally, the reverse is true that if a message is received with a process name, P, and a timestamp, T, that differs from Q's time by more B, then Q will suspect P. The specification of this oracle is the same check being performed currently in the Kerberos protocol. As such, replacing these current checks with the specification of the oracle does not change the correctness of this protocol. Figures 3.5, 3.6, and 3.7 show the Kerberos protocol augmented with the oracle; note that C is not affected by adding the oracle.

```
process C

const
    KS, TS
var
    startc    :  boolean,  {initially, startc = false}
    keyc      :  array[PName] of integer,
                  {initially, keyc[KS] = K_{C,KS} ∧
                   keyc[TS] = ⊥  ∧  keyc[S] = ⊥}
    nc        :  integer,   {initially, nc = ⊥}
    x,y,z     :  integer

begin
~startc  →
    nc, startc := NNC, true;
    send krqst(C, TS, nc) to AS
[]
rcv krply(x, y) from AS →
    (x, z) := DCR(keyc[KS], x);
    if z ≠ nc →  startc, nc := false, ⊥
    [] z = nc  →
        keyc[TS], nc := x, NNC;
        send trqst(NCR(keyc[TS], TIM), y, S, nc) to AS
    fi
[]
rcv trply(x, y) from AS →
    (x, z) := DCR(keyc[TS], x);
    if z ≠ nc → startc, nc, keyc[TS] := false, ⊥, ⊥
    [] z = nc  →
        keyc[S], nc := x, ⊥;
        send rqst(NCR(keyc[S], TIM), y) to S
    fi
end
```

Figure 3.5: Client with Oracle

```
process AS
const
    KS, TS
inp
    keyks           :  array[PName] of Keys,
                        {keyks[TS] = K_{KS,TS} ∧
                         keyks[C] = K_{C,KS} ∧ keyts[S] = ⊥}
    keyts           :  array[PName] of Keys,
                        {keyts[KS] = K_{KS,TS} ∧
                         keyts[C] = ⊥ ∧ keyts[S] = K_{TS,S}}
    L               :  integer,
    D               :  integer
var
    keyas           :  integer,  {initially, keyas = ⊥}
    A               :  integer,
    u,v,w,x,y,z     :  integer

begin
rcv krqst(x, y, z) →
    if keyks[x] = ⊥ ∨ keyks[y] = ⊥ → skip
    [] keyks[x] ≠ ⊥ ∧ keyks[y] ≠ ⊥ →
        keyas := Random;
        send krply(NCR(keyks[x], (keyas, z)),
           NCR(keyks[y], (x, TIM + L, keyas))) to x
        keyas := ⊥
    fi
[] rcv trqst(w, x, y, z) →
    (u, v, x) := DCR(x, keyts[K]);
    if (Suspect(KS, v, 0)) → skip
    [] (~Suspect(KS, v, 0)) →
        A := DCR(w, x);
        if (Suspect(u, A, D)) → skip
        [] (~Suspect(u, A, D)) →
            keyas := Random;
            send trply(NCR(x,(keyas, z)),
              NCR(keyts[y], (u, TIM + L, keyas))) to u
            keyas := ⊥
        fi
    fi
end
```

Figure 3.6: Authentication Server with Oracle

```
process S
const
    KS, TS
inp
    D           :  integer
var
    auths       :  boolean ,  {initially , auths = false}
    keys        :  array[PName] of integer ,
                    {initially , keys[KS] = ⊥ ∧
                     keys[TS] = K_{TS,S} ∧ keys[C] = ⊥}
    A           :  integer ,
    v,w,x,y,z   :  integer

begin
rcv rqst(v, w) −>
    w, x, y := DCR(keys[TS], w);
    if (Suspect(TS, x, 0)) −> skip
    [] (∼Suspect(TS, x, 0)) −>
        A := DCR(y, v);
        if (Suspect(u, A, D) ∨ keys[C] ≠ ⊥) →
            skip
        [] (∼Suspect(u, A, D) ∧ keys[C] = ⊥) →
            keys[C], auths := y, true;
        fi
    fi
end
```

Figure 3.7: Server with Oracle

CHAPTER 4

KERBEROS WITH INTRUDERS

The assumptions that Kerberos makes with respect to time are quite strict. How do the guarantees of Kerberos change if the protocol is deployed in an environment that does not adhere to such strong requirements? In [Gon92], Li Gong presents the *suppress-replay* attack that expresses a vulnerability or insecurity in the Kerberos protocol. Suppress-replay exploits Kerberos's requirement on synchronized clocks. In many networks, this requirement is hard, if not impossible, to maintain. The attack addresses a vulnerability created when the clock of C advances ahead of real time and C sends a message to S. Figure 4.1 shows a message being sent before the fault, M_1, the fault occurring, and a message being sent after the clock of C has drifted ahead 4 time units. In this example, we are considering 4 to be outside the window of acceptable drift. The vertical lines mark specific moments in real time, namely T_1 and T_2. So M_1 is sent at real time T_1, and M_2 is sent at real time T_2.

The attack that Gong composes exploits the last message in Kerberos, sent from C to S. This message contains a freshly generated timestamp from C. In this example, when C incorporates a timestamp in message M_1, the clock of C has the same value as real time and the timestamp T_1 is included in message M_1. When

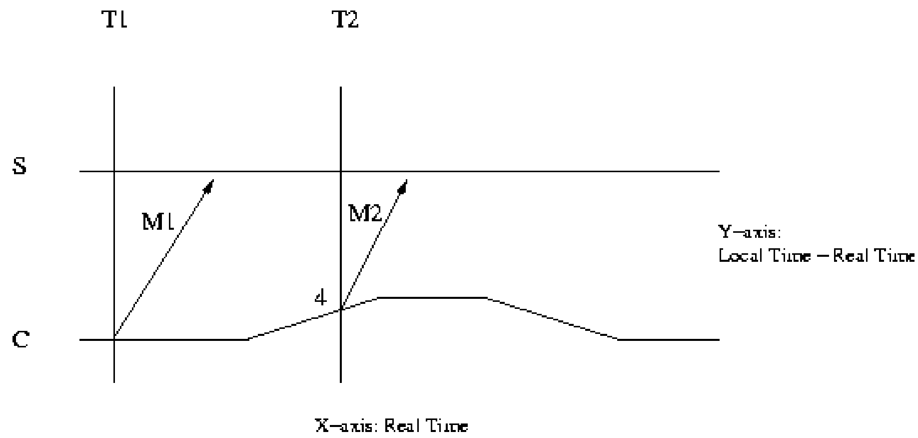

Figure 4.1: Clock Asynchrony

message M_2 is constructed, the clock at C has drifted ahead. The timestamp included in message M_2 is $T_2 + 4$.

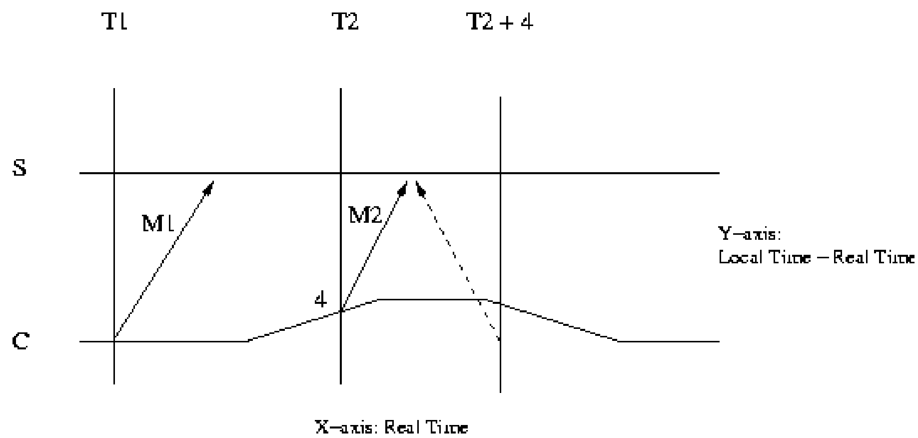

Figure 4.2: Future Timestamp

Gong considers two scenarios, one where the message reaches S and one in which the message is suppressed. In the first case, when the message arrives at S,

it appears to have been generated in the future (Figure 4.2). In this case, S will deny access to the client, since the timestamp is not inside the acceptable window. If the message never arrives at S, C is trivially not authenticated by S. In either case, the message generated contains correct information, it is simply not valid at the current moment in real time.

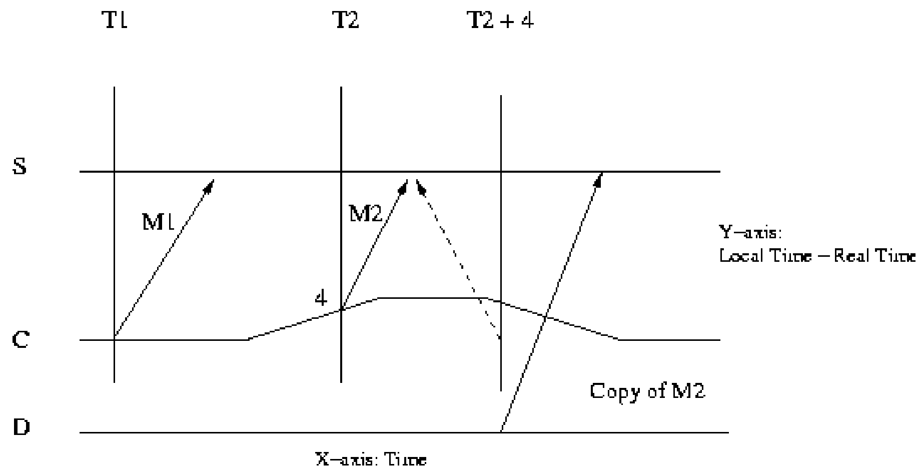

Figure 4.3: Replayed Message

The vulnerability arises from the creation of a message that is not currently valid, but will be valid at some point in the future. Gong posits a simple intruder, I, that either copies or suppresses the message with a future stamp, M_2. I holds onto the message and sends the message at a future time when the message is valid, Figure 4.3. The intruder still cannot see the information inside the message. Thus, the intruder must guess when the timestamp inside the message becomes valid. This message will be indistinguishable from a valid message created by C and S will authenticate I as C when sent at the appropriate time, violating one of the goals of Kerberos.

4.1 Intrusion States

Gong describes two sets of intruders. The weaker set of intruders is capable of copying messages from the channel $ch.C.S$ and inserting messages into channel $ch.C.S$. The stronger set of intruders has the ability to remove and insert messages on channel $ch.C.S$. Each of element of these sets can be realized in AP notation. For instance in one realization, the intruder might remove only the last message in the channel; in another, the intruder might remove only messages at the front of the channel. We label the weaker set of intruders as the set WI, the stronger as the set SI. We limit the intruders in both sets to removing and inserting a finite number of messages from the channel. Thus, we do not consider denial of service in any form.

In order to represent this fault scenario for Kerberos, we refine the state by making the following substitutions. We use v and nv to represent valid and invalid messages respectively.

$$A = \mathbf{krqst}(nc)\#ch.C.AS + \mathbf{trqst}(nc)\#ch.C.AS$$

A is the number of **krqst** and **trqst** messages whose nonces are equal to the value of variable nc in C.

$$B = \mathbf{krply}(nc)\#ch.AS.C + \mathbf{trply}(nc)\#ch.AS.C$$

B is the number of **krply** and **trply** messages whose nonces are equal to the value of variable nc in C.

$$D = \mathbf{rqst}(v)\#ch.C.S$$

D is the number of messages in the channel from C to S with valid timestamps sent from process C.

$$E = \mathbf{rqst}(iv)\#ch.C.S$$

E is the number of messages in the channel from C to S with invalid timestamps sent from process C.

$$F = \mathbf{rqst}(v)\#ch.C.S$$

F is the number of messages in the channel from C to S with valid timestamps sent from process D.

$$G = \mathbf{rqst}(iv)\#ch.C.S$$

G is the number of messages in the channel from C to S with invalid timestamps sent from process D.

We make substitutions into the previous state predicates. We also adjust the states to represent the fault that may be injected in the system, namely that C's clock drifts out of the acceptable bound. This affects only one state of the protocol, **q.5**. This state now splits into two states, **q.5'** and **q.6**. Previously, C created only valid timestamps with **q.5'** representing a message with a valid timestamp created by C. The new state, **q.6**, represents a message with an invalid timestamp or the occurrence of a fault in C. The resulting state diagram is picture in Figure 4.4. For all other states, we explicitly substitute the above predicates. The resulting states are listed in Appendix B.

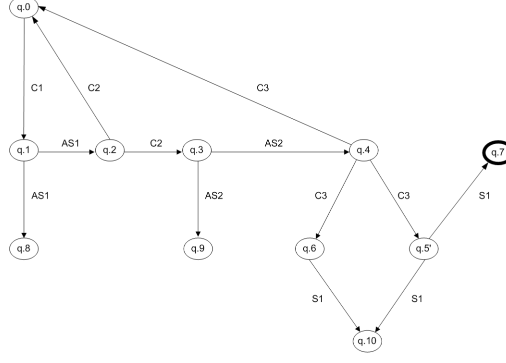

Figure 4.4: State Transition Diagram

q.5': $A = B = E = F = G = 0 \wedge D = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.6: $A = B = D = F = G = 0 \wedge \quad \wedge E = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc = \wedge \sim auths \wedge Z$

Composing intruders with the protocol expands the reachable states for the protocol and creates a set of unsafe states. These states mimic the safe states, changing only the number of messages in channel $ch.C.S$ inserted from the intruder (F and G). Since the intruder may insert messages with valid timestamps as well as invalid timestamps, each safe state, **q.i**, corresponds to three unsafe states: one in which the intruder has inserted valid messages, **q.i.f**, one in which the intruder has inserted invalid messages, **q.i.g**, and one in which the intruder has inserted both valid and

invalid messages, **q.i.fg**. Here e, f, and ef, correspond to F $>$ 0, G $>$ 0 or (F $>$ 0 and G $>$ 0) respectively. Thus, safe state **q.0** corresponds to unsafe states **q.0.f**, **q.0.g**, and **q.0.fg**. In the state diagrams, we see each safe state and the three corresponding unsafe states connected in a small diamond. The transitions among states are indicated by the name of the process executing an action that creates the transition, S represents the Server action and D represents an Adversary action. Due to space considerations, the transitional actions are shown only in the smaller figure, not in the figures containing all the states. The intruder actions (inserting messages) create transitions from safe states to unsafe states. The transitions among unsafe states can be either actions of the intruder (inserting messages) or actions of the server (retrieving intruder messages). The transitions from unsafe states to safe states occur upon execution of the server action (retrieving intruder messages). Figure 4.5 displays the state transition diagram for the diamond associated with **q.0**; the states in the diagram are listed below.

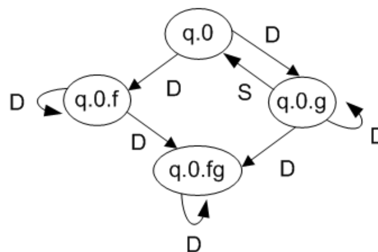

Figure 4.5: State Diamond

$$\textbf{q.0:}\ A = B = D = E = F = G = 0 \wedge$$
$$keyc[TS] = \bot \wedge keyc[S] = \bot \wedge keys[C] = \bot \wedge$$
$$startc = false \wedge auths = false \wedge Z$$

$$\textbf{q.0.f:}\ A = B = D = E = G = 0 \wedge F > 0 \wedge$$
$$keyc[TS] = \bot \wedge keyc[S] = \bot \wedge keys[C] = \bot \wedge startc = false \wedge Z$$

$$\textbf{q.0.g:}\ A = B = D = E = F = 0 \wedge G > 0 \wedge$$
$$keyc[TS] = \bot \wedge keyc[S] = \bot \wedge keys[C] = \bot \wedge startc = false \wedge Z$$

$$\textbf{q.0.fg:}\ A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$$
$$keyc[TS] = \bot \wedge keyc[S] = \bot \wedge keys[C] = \bot \wedge startc = false \wedge Z$$

Each diamond contains unsafe transitions from states with valid intruder messages, **q.i.f** and **q.i.fg**, to states in the **q.7** diamond Figure 4.6. All states in the **q6** diamond have accepted a valid message and set $auths$ to true. The transitions among the diamonds demonstrate the vulnerability created by the intruder inserting valid message into the channel. These transitions model S authenticating a valid intruder message.

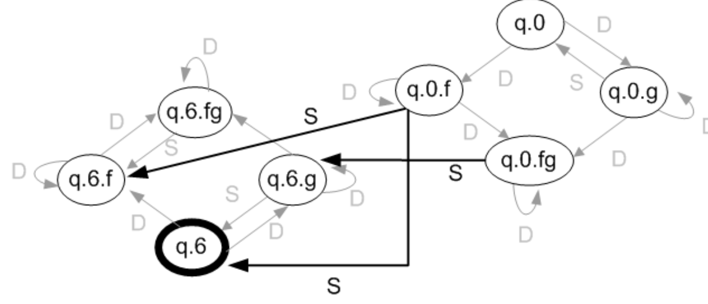

Figure 4.6: Unsafe Transitions

4.2 Vulnerability of Kerberos

We consider the composition of the Kerberos Protocol, P, with intruders, WI and SI, described in the previous section. The properties below are defined for a general intruder, D. We divide the reachable state predicates of P into the set of safe or valid states, **Y**, and the set of unsafe states, **U**. The set of critical variables, **V**, in P (those that need protection from D), contains one element, $auths$.

$$\mathbf{Y} = \bigcup_{i=0}^{10} q.i$$

$$\mathbf{U} = \bigcup_{i=0}^{10} \{q.i.f, q.i.g, q.i.fg\}$$

$$\mathbf{Q} = \mathbf{Y} \cup \mathbf{U}$$

$$\mathbf{V} = \{auths\}$$

Security to an intruder can be established by using three concepts of stabilization theory: closure, convergence, and protection [Gou01],[AG93]. When composed with an intruder, a protocol is secure against the intruder if these three conditions are met and insecure if protection is violated.

1. *Closure*: The safe states, **Y**, are closed under the execution of any action of P; the reachable states, **Q**, are closed under the execution of any action of P or any action of D.

2. *Convergence*: Starting in any reachable state, any infinite execution of the actions of P lead to safe states.

3. *Protection*: No variable in **V** is written in any transition from a unsafe state to a safe state.

We consider the security of PW, the composition of the Kerberos protocol, P, and the weaker set of intruders, WI.

PW **is closed.** From Figure 4.4, we see that the safe states, **Y**, are closed under any action of P. The reachable states are closed under any action of PW. Executing any action in PW results in a transition either inside a diamond or across diamonds.

PW **converges.** No intruder messages are added to the channels: $ch.C.AS$, $ch.AS.C$, or $ch.S.C$. Each intruder message added to $ch.C.S$ is eventually received at S. An intruder can insert only a finite number of messages. Thus, for any infinite run of the protocol, starting at an unsafe state $u \in U$ leads to a safe state $x \in X$ where no intruder messages exist in $ch.C.S$.

PW **is NOT secure.** The Kerberos protocol is vulnerable when composed with an intruder WI. When WI inserts a message with a valid timestamp and valid lifetime into $ch.C.S$, this message is indistinguishable from a valid message sent from C . Thus, when S receives the message, S will update the *auths* variable, creating a vulnerability in Kerberos. Start in any unsafe state that contains valid

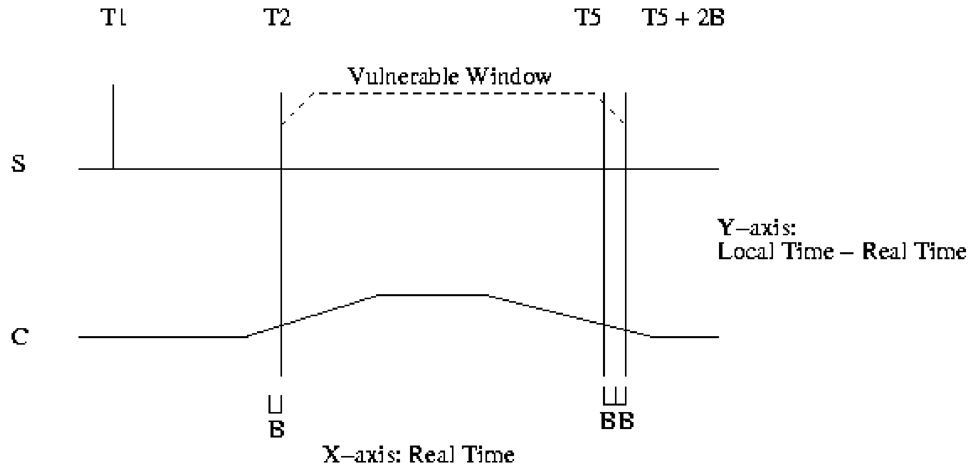

Figure 4.7: Vulnerable Window 1

intruder messages, **q.i.f** or **q.i.fg**. Action 1 of S may receive a valid message from the intruder. Since the message is valid, S will set the *auths* variable and transition to a state in the **q.7** diamond. Once *auths* is set, its value never changes. Thus, once entering the **q.7** diamond, no transition leads away from this diamond. Eventually, all intruder messages will be received by S, and the resulting state in the **q.7** diamond is **q.7**.

4.3 The Window of Vulnerability

Kerberos requires that clocks remain synchronized within a bound, B. The result is that messages arriving at a principal will be accepted within a $2B$ window (*i.e. current time* $\pm$ B). When a clock in the Kerberos system drifts out of the acceptable bound, it creates a window of vulnerability. The size of this window will be the duration for which the clock is outside of the bound plus an additional

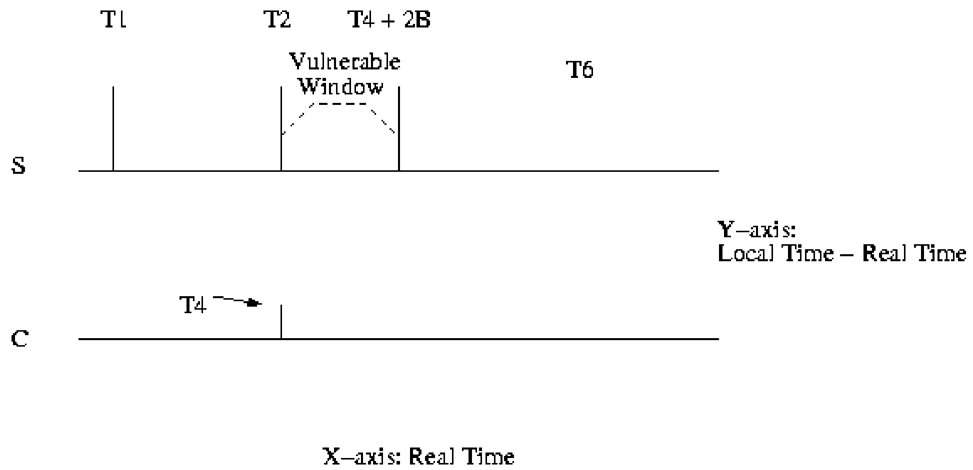

Figure 4.8: Vulnerable Window 2

$2B$ beyond the greatest time encountered. The following figures (4.7, 4.8, and 4.9) demonstrate the vulnerable windows created by three different kinds of faults.

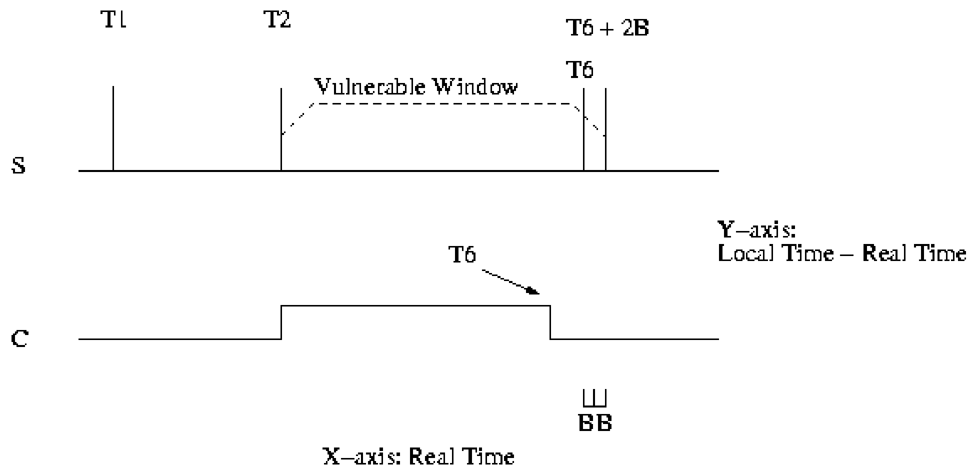

Figure 4.9: Vulnerable Window 3

In Figure 4.7, we see C's clock drifting above the bound, remaining ahead from T2 to T5, and slowing back down to keep pace with real time. This type of drift

creates a vulnerability both when C has drifted above the bound (T2 to T5), as well as an additional $2B$ time units. The greatest clock value that C holds outside the bound is T5. Thus, the greatest timestamp that could be included in a message will be T5. A message with timestamp T5 will be accepted by S until time T5 + $2B$. The vulnerable window created by this clock drift is T2 to T5+$2B$.

In figures 4.8 and 4.9 notice that the vertical lines extend only to the horizontal line representing S's clock. This is to avoid confusion between the C's clock jumping values and moments in real time. Since C's clock is jumping up in both these figures, the vertical slices extend only partially down the figure. For each moment that C's clock is outside the allowable bound, a vulnerability is created in the system for $2B$ time units. Figure 4.8 shows C's clock jumping above the bound for one instance in time. When C's clock jumps ahead at time T2, it contains the value T4. In this scenario, the vulnerable window spans from the out of bounds moment until $2B$ plus the largest time held at C, T4+$2B$.

Figure 4.7, shows C's clock jumps ahead of real time at T2 and remaining ahead for an extended time period. The greatest value that C's clock has before resetting is T6. If C's clock drifts out of the acceptable bound at time T2 and has a largest value of T6, a window of vulnerability is created from T2 until (T6+$2B$). Notice here, as in the previous example, the window extends $2B$ past the largest time held at C. C may reset sometime before T6, however the $2B$ is appended to the greatest timestamp held, not the time at which C reset.

4.4 Modifying the Oracle

Kerberos requires a stronger oracle in order to thwart the suppress-replay attacks. Since we have decoupled timing concerns from authentication concerns, we can adapt the abstract properties of the oracle. Kerberos augmented with a stronger oracle can then provide more guarantees on the security provided. We strengthen the oracle first to impose the restrictions of "loosely synchronized" clocks. We then show that augmenting Kerberos with this first strengthening does not thwart suppress-replay attacks. Next, we create a stronger oracle specification that is able to thwart the suppress-replay intruder.

The original specification ($Oracle_1$) created an abstraction that captured the check in the Kerberos protocol for a process, Q . This specification did not place any restrictions on the synchronization of clocks at different processes.

$$\textbf{Suspect(PName P, Timestamp T, Bound B)} \leftrightarrow \ |time_Q - T| > B$$

We modify this specification to indicate that not only should the timestamp, T, in the message be accurate, but also the clocks at two processes should be within a bound of each other.

$$\textbf{Suspect(PName P, Timestamp T, Bound B)} \leftrightarrow$$

$$|time - T| > B \vee |time_Q - time_P| > B$$

$Oracle_2$ identifies invalid timestamps within messages as well as durations in which the clock at a process has drifted out of the acceptable bound. In the timing diagrams, this oracle identifies a window starting when the clock drifts beyond the bound and ending when the clock drift returns inside the bound. The suppress-replay intruder creates a window of vulnerability for an additional $2B$ time units.

In order to capture this window, we must track when the clocks drift beyond the bound as well as the $2B$ time units in the future for which a vulnerability is created. The second strengthening ($Oracle_3$) captures the window of vulnerability created by the suppress-replay intruder.

$$\textbf{Suspect(PName p, Timestamp T, Bound B)} \leftrightarrow$$

$$|time_Q - T| > B \vee |time_Q - time_P| > B \vee$$

$$\exists x : 0 \ldots 2B : |time_{Q,x} - time_{P,x}| > B$$

In this oracle, a process Q suspects a process P if any of the following conditions hold. The time at Q differs from the timestamp by more that a bound, B. The time at process P and the time at process Q differ by more than B. Or, the time at process P has differed from the time at process Q by more that B at some point within the last $2B$ time units.

4.5 Correctness

PW **augmented with** $Oracle_3$ **is closed.** See Section 4.2.

PW **augmented with** $Oracle_3$ **converges.** See Section 4.2.

PW **augmented with** $Oracle_3$ **is secure.** The Kerberos protocol is secure when composed with WI. When WI inserts a message with a valid timestamp and valid lifetime into $ch.C.S$, although the message is valid, the oracle will currently suspect C. Thus, when S receives the message, S will discard the message without updating the $auths$ variable. Additionally, if WI inserts an invalid message, this too will be discarded by S. Thus, the $auths$ variable is never set upon receipt of a message from WI. Hence, no transition from an unsafe state to a safe state effects the variable $auths$.

CHAPTER 5

RELATED WORK

We introduce related work in three areas: designing authentication protocols, reasoning about authentication protocols, and detection oracles. We briefly describe these works and provide a comparison to the results presented in this thesis.

In practical systems, synchronized clocks can increase performance [Lis91, Gon93, DS81]. Existing algorithms, such as the Network Time Protocol (NTP) [Milxt], provide mechanisms for the probabilistic synchronization of clocks. While clocks can be synchronized with a very high degree of confidence, depending this synchronization for correctness is problematic [Lis91, Gon92, BM91]. These problems arise when the performance enhancements side effects correctness. While many have described the dangers associated with relying on clock synchronization, to our knowledge none have used oracles to characterize this space.

Many techniques exist for formally verifying authentication protocols, such as modal logics, state machines, and theorem provers. Catherine Meadow surveys these techniques in [Mea94]. Modal logics address belief and knowledge that can be derived in a distributed system. Perhaps the most well-known is BAN logic [BAN96]. The goal of authentication, as well as the initial assumptions are cast into sets of beliefs. After each message is received, a new set of beliefs is derived by applying

inference rules to the most recent set of beliefs. If the goal can be reached through such applications, then the protocol has been proven correct. Discussions of this area can be found in [SC01, Syv92].

State machine approaches are based on the work of Dolev and Yao [DY81] and of Dolev, Even and Karp [DEK82]. These approaches analyze whether the reachable states of the program contain unsafe states. State machine approaches address the states themselves, while the concepts of stabilization discuss transitions among these states.

Bella and Paulson [Pau98, Bel94] have explored the use of theorem provers to inductively verify authentication protocols and have applied this approach to Kerberos [BP97, BP98b, BP98a]. None of their approaches consider time and how time affects correctness.

Syverson [Syv93] augments BAN logic with temporal logic. This augmented logic still does not consider synchronization parameters necessary for correctness.

Chandra and Toueg [CT96] introduced the use of oracles for detecting faults in distributed systems. Their approach targeted solving the problem of consensus in different distributed models. Their detectors create abstractions of different distributed models. These detectors can then be considered to determine the weakest parameters in which a problem can be solved. Chandra and Toueg address persistent failures. While others have created detection oracles for transient failures [BDDT98], none have created detection oracles for clock drift.

CHAPTER 6

CONCLUSIONS

This thesis makes two contributions. We create oracles that encapsulate network timing parameters decoupling them from the correctness of Kerberos, and we discuss the composition of Kerberos with different oracle specifications. We compose Kerberos with a modified oracle and show how this composition thwarts adversaries that Kerberos alone could not.

Currently, the Kerberos has a direct dependence on network timing parameters. Thus, these timing parameters must be known to exist in order to dispatch the correctness of the Kerberos protocol. We decouple the correctness of Kerberos from network timing parameters by encapsulating the timing parameters in an oracle. This allows for the correctness of Kerberos to be dispatched relative to abstract properties of the oracle. These properties can be realized by many different networks without requiring new proof obligations. In addition, modifications to the timing parameters can be made independent of the protocol.

We model Kerberos using Abstract Protocol Notation, and assess the security properties of Kerberos using three concepts of stabilization theory: closure, convergence, and protection. We demonstrate the insecurity created by composing Kerberos with Gong's weak and strong adversaries. We then discuss the modifications

of the oracle necessary to thwart these adversaries. We show that augmenting Kerberos with a modified oracle provides security against Gong's adversaries. Finally, we discuss the related work in this area.

APPENDIX A

A REALIZATION FOR RANDOM WITHOUT $\perp$

Our realization of **Random** simply uses **random** but excludes special symbols. The following example considers the symbol $\perp$ to be defined as a specific integer constant (the same assumption made in the realization of Kerberos). Figure A.1, defines **Random** using AP notation. In this figure we assign x a **Random** value that is not equal to $\perp$.

```
process Random

var
    x  :  integer  {initially , x = ⊥}

begin

true →
    do x = ⊥ →
        x := random
    od
end
```

Figure A.1: Random

APPENDIX B

STATES

B.1 Safe States

$$\textbf{q.0:}\ A = B = D = E = F = G = 0 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$\sim startc \wedge \sim auths \wedge Z$$

$$\textbf{q.1:}\ B = D = E = F = G = 0 \wedge A = 1 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge \sim auths \wedge Z$$

$$\textbf{q.2:}\ A = D = E = F = G = 0 \wedge B = 1 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge \sim auths \wedge Z$$

q.3: $B = D = E = F = G = 0 \wedge A = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.4: $A = D = E = F = G = B = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.5': $A = B = E = F = G = 0 \wedge D = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.6: $A = B = D = F = G = E = 1 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.7: $A = B = D = E = F = G = 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = K_{C,S} \wedge$

$startc \wedge auths \wedge Z$

$$\textbf{q.8:}\ A = B = D = E = F = G = 0 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge \sim auths \wedge Z$$

$$\textbf{q.9:}\ A = B = D = E = F = G = 0 \wedge$$
$$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$$
$$startc \wedge \sim auths \wedge Z$$

$$\textbf{q.10:}\ A = B = D = E = F = G = 0 \wedge$$
$$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$$
$$startc \wedge \sim auths \wedge Z$$

B.2 Safe and Unsafe States

$$\textbf{q.0.f:}\ A = B = D = E = G = 0 \wedge F > 0 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge \sim startc \wedge \sim auths \wedge Z$$

$$\textbf{q.0.g:}\ A = B = D = E = F = 0 \wedge G > 0 \wedge$$
$$keyc[TS] = keyc[S] = keys[C] = \perp \wedge \sim startc \wedge \sim auths \wedge Z$$

q.0.fg: $A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge \sim startc \wedge \sim auths \wedge Z$

q.1.f: $B = D = E = G = 0 \wedge A = 1 \wedge F > 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.1.g: $B = D = E = F = 0 \wedge A = 1 \wedge G > 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.1.fg: $B = D = E = 0 \wedge A = 1 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.2.f: $A = D = E = 0 \wedge B = 1 \wedge F > 0 \wedge G = 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.2.g: $A = D = E = F = 0 \wedge B = 1 \wedge G > 0 \wedge$

$keyc[TS] = keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.2.fg: $A = D = E = 0 \land B = 1 \land F > 0 \land G > 0 \land$

$keyc[TS] = keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.3.f: $B = D = E = 0 \land G = 0 \land A = 1 \land F > 0 \land$

$keyc[TS] = K_{C,TS} \land keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.3.g: $B = D = E = F = 0 \land A = 1 \land G > 0 \land$

$keyc[TS] = K_{C,TS} \land keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.3.fg: $B = D = E = 0 \land A = 1 \land F > 0 \land G > 0 \land$

$keyc[TS] = K_{C,TS} \land keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.4.f: $A = D = E = G = 0 \land B = 1 \land F > 0 \land$

$keyc[TS] = K_{C,TS} \land keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.4.g: $A = D = E = F = 0 \land B = 1 \land G > 0 \land$

$keyc[TS] = K_{C,TS} \land keyc[S] = keys[C] = \bot \land startc \land \ \sim auths \land Z$

q.4.fg: $A = D = E = 0 \wedge B = 1 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge startc \wedge \sim auths \wedge Z$

q.5.f: $A = B = E = G = 0 \wedge D = 1 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.5.g: $A = B = E = F = 0 \wedge D = 1 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.5.fg: $A = B = E = F = 0 \wedge D = 1 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.6.f: $A = B = D = E = 1 \wedge G = 0 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.6.g: $A = B = D = 0 \wedge F = 0 \wedge E = 1 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.6.fg: $A = B = D = 0 \wedge E = 1 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.7.f: $A = B = D = E = G = 0 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = \perp \wedge keys[C] = K_{C,S} \wedge$

$startc \wedge auths \wedge Z$

q.7.g: $A = B = D = E = F = 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = \perp \wedge keys[C] = K_{C,S} \wedge$

$startc \wedge auths \wedge Z$

q.7.fg: $A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = \perp \wedge keys[C] = K_{C,S} \wedge$

$startc \wedge auths \wedge Z$

q.8.f: $A = B = D = E = G = 0 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.8.g: $A = B = D = E = F = 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.8.fg: $A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = K_{C,S} \wedge keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.9.f: $A = B = D = E = G = 0 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.9.g: $A = B = D = E = F = 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.9.fg: $A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = \perp \wedge$

$startc \wedge \sim auths \wedge Z$

q.10.f: $A = B = D = E = 0 \wedge G = 0 \wedge F > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = K_{C,S} \wedge$

$startc \wedge \sim auths \wedge Z$

q.10.g: $A = B = D = E = F = 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = keys[C] = K_{C,S} \wedge$

$startc \wedge \sim auths \wedge Z$

q.10.fg: $A = B = D = E = 0 \wedge F > 0 \wedge G > 0 \wedge$

$keyc[TS] = K_{C,TS} \wedge keyc[S] = \perp \wedge keys[C] = K_{C,S} \wedge$

$startc \wedge \sim auths \wedge Z$

BIBLIOGRAPHY

[AG93] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, nov 1993. Special Issue on Software Reliability.

[BAN96] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In William Stallings, editor, *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[BDDT98] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC'98)*, number 1499, pages 62–74, Andros, Greece, 1998. Springer-Verlag.

[Bel94] Giampaolo Bella. Inductive verification of cryptographic protocols, 2000 1994.

[BM91] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991. USENIX.

[BP97] G. Bella and L. Paulson. Using isabelle to prove properties of the kerberos authentication system. In H. Orman and C. Meadows, editors, *Workshop on Design and Formal Verification of Security Protocols. DIMACS*, 1997.

[BP98a] Giampaolo Bella and Lawrence C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proceedings of the 5th European Symposium on Research in Computer Security*, pages 361–375, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag LNCS 1485.

[BP98b] Giampaolo Bella and Lawrence C. Paulson. Mechanising BAN kerberos by the inductive method. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV'98, 10th Conference on Computer Aided Verification*, pages 416–427, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag LNCS 1427.

[CDEGR90] G. A. Champine, Jr. D. E. Geer, and W. N. Ruh. Project Athena as a distributed computer system. *IEEE Communications Magazine*, 23(9):40–51, September 1990.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[DEK82] D. Dolev, S. Even, and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55:57–68, 1982.

[DS81] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *CACM*, 24(7):533–535, August 1981.

[DY81] D. Dolev and A. Yao. On the security of public-key protocols. In *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science*, pages 350–357, 1981.

[Gon92] Li Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, 1992.

[Gon93] Li Gong. Lower bounds on messages and rounds for network authentication protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 176–183, November 1993.

[Gou98] Mohomad G. Gouda. *Elements of Network Protocol Design.* John Wiley & Sons, INC, 1998.

[Gou01] Mohamed G. Gouda. Elements of security: Closure, convergence, and protection. *Information Processing Letters*, 77(2–4):109–114, February 2001.

[KNT91] John T. Kohl, B. Clifford Neuman, and Theodore Y. Ts'o. The evolution of the Kerberos authentication service. In *Proceedings of the Spring 1991 EurOpen Conference*, 1991.

[KPS02] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a PUBLIC World*. Prentice Hall PTR, 2nd edition, 2002.

[Lis91] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 1–9, 1991.

[Mea94] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology.* LNCS, Springer-Verlag, 1994.

[Milxt] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992 http://www.networksorcery.com/enp/rfc/rfc1305.txt.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, 1987. http://www.mit.edu/afs/athena/astaff/-project/kerberos/www/papers.html.

[Neu] B. Clifford Neuman. The Kerberos Network Authentication Service (V5). Internet request for comment RFC, Internet Engineering Task Force.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[SC01] Paul Syverson and Iliano Cervesato. The logic of authentication protocols. *Lecture Notes in Computer Science*, 2171:63+, 2001.

[SNS88] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988. USENIX Association.

[Syv92] Paul F. Syverson. Knowledge, belief, and semantics in the analysis of cryptographic protocols. *Journal of Computer Security*, 1(3-4):317–334, 1992.

[Syv93] Paul F. Syverson. Adding time to a logic of authentication. In *Proceedings of the 10st ACM conference on Computer and Communications Security*, pages 97–101, 1993.

[Tun99] Brian Tung. *Kerberos: a network authentication system.* Addison-Wesley, Reading, Mass., 1999.