Automatic Verification of Transmission Control Protocol Using NuSMV

Jingjing Lu **Yuxiang Zhu**

March 27, 2000

Abstract

In this report we construct a model of the TCP state machine and verify it using NuSMV. Also we check our model of TCP for some denial-of-services attacks. Finally we address the issue whether NuSMV is a proper model checker for complex systems like TCP.

1. Introduction

In this section, we explain the ideas of TCP and give a brief introduction to the model checker NuSMV; finally, we introduce the basic concepts of TCP attacks.

1.1 A Brief Introduction to TCP

TCP (Transmission Control Protocol) is a very important and well-known network-level protocol. Although TCP is always mentioned as part of the TCP/IP Internet protocol suite, it is an independent, general-purpose protocol that can be adapted for use with other delivery systems. TCP has been so popular that one of the International Organization for Standardization's open systems protocols, TP-4, has been derived from it.

TCP is a very complex, high level protocol. It specifies the format of the data and acknowledgments that two computers exchange to achieve a reliable transfer, as well as the procedures the computers use to ensure that the data arrives correctly. This reliable stream delivery service guarantees to deliver a stream of data sent from one machine to another without duplication or data loss. TCP ensures the reliability using positive acknowledgment with retransmission. This technique requires a recipient to communicate with the source, sending back an acknowledgment (ACK) message as it receives data. The sender keeps a record of each packet it sends and waits for an acknowledgment

before sending the next packet. The sender also starts a timer when it sends a packet and retransmits this packet if the timer expires before an acknowledgment arrives.
Figure 1 shows how the simplest positive acknowledgement protocol transfers data.

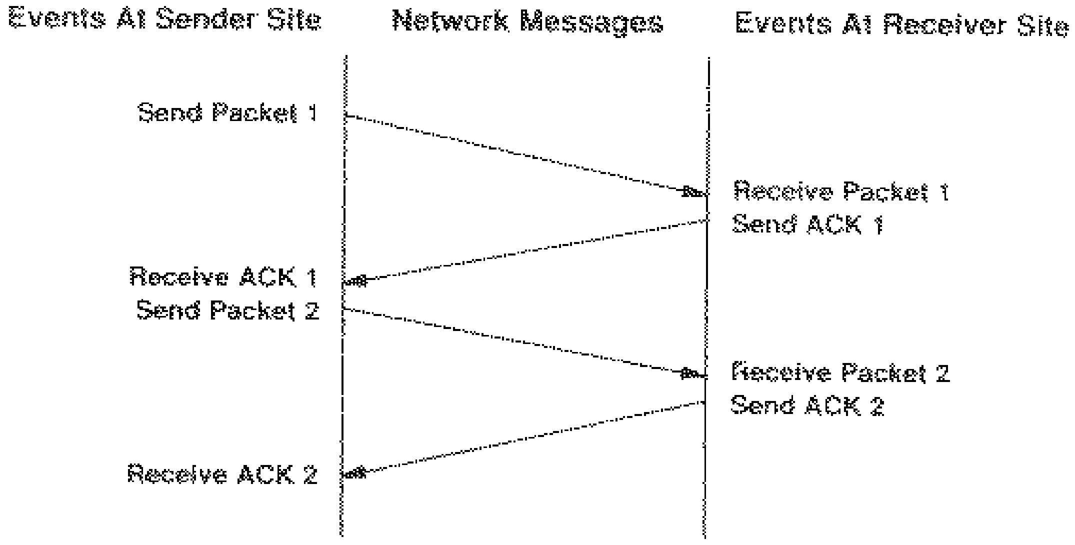

Figure 1 A protocol using positive acknowledgment with retransmission in which the sender awaits an acknowledgment for each packet sent. Vertical distance down the figure represents increasing time and diagonal lines across the middle represent network packet transmission. ([2])

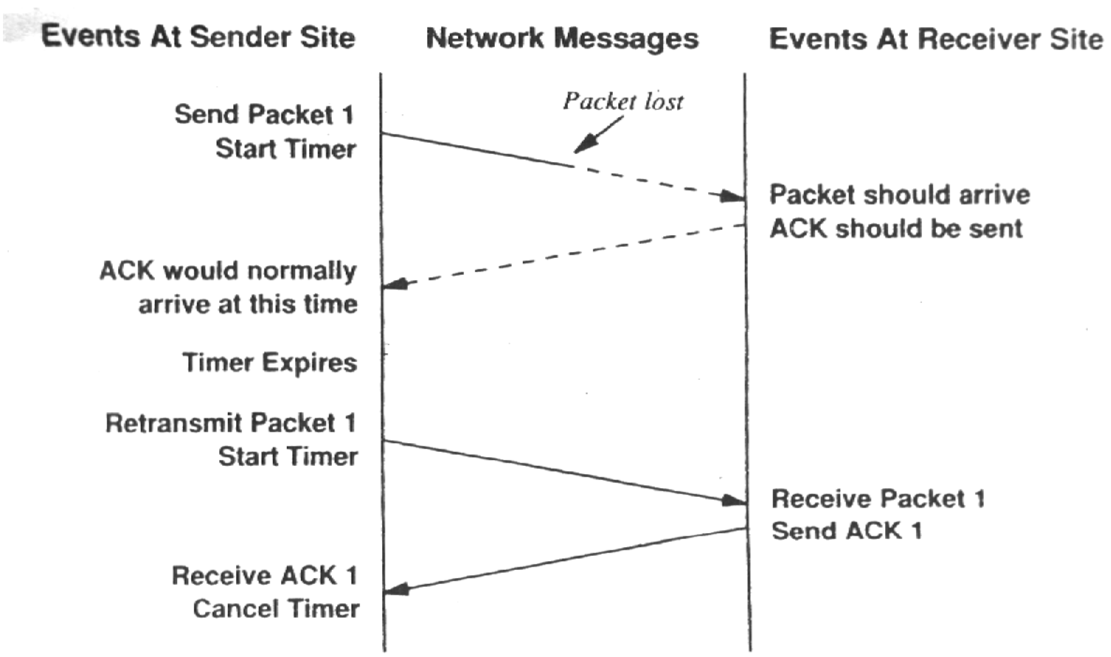

Figure 2 Timeout and retransmission that occurs when a packet is lost. The dotted lines show the time that would be taken by the transmission of a packet and its acknowledgment, if the packet was not lost. ([2])

Figure 2 uses the same format diagram as Figure1 to show what happens when a packet is lost or corrupted.

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since each of octets is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. This mechanism allows for straightforward duplicate detection in the presence of retransmission.

To establish a connection, TCP uses a three-way handshake. In the simplest case, the handshake proceeds as shown in Figure 3.

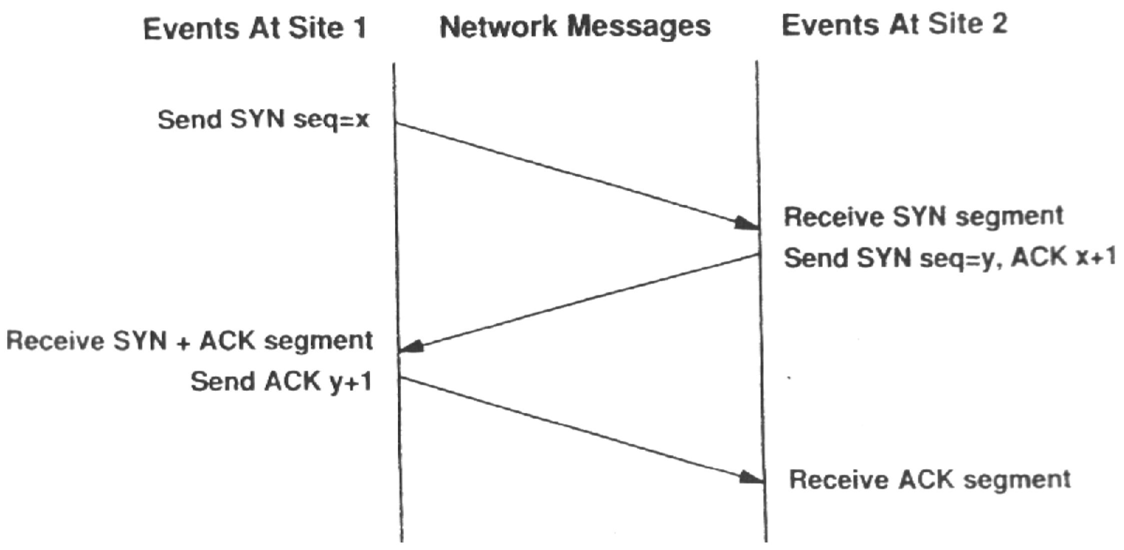

Figure 3 The sequence of messages in a three-way handshake. SYN segments carry initial sequence number information. ([2])

The three-way handshake is both necessary and sufficient for correct synchronization between the two ends of the connection. Since TCP builds on unreliable packet delivery services, messages can be lost, delayed, duplicated or delivered out of order. Thus, the protocol must use a timeout mechanism and retransmit lost requests. Trouble arises if retransmitted and original requests arrive while the connection is being established, or if retransmitted requests are delayed until after a connection has been established, used or terminated. A three-way handshake (plus the rule that TCP ignores additional requests for connection after a connection has been established) solves these problems.

The following is a list of states during the lifetime of a connection: **LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT**, and the fictional state **CLOSED. CLOSED** is fictional because it represents the state when there is no TCB (Transmission Control Block), and therefore, no connection.

The states have the following meaning:

- **LISTEN** - represents waiting for a connection request from any remote TCP and port.
- **SYN-SENT** - represents waiting for a matching connection request after having sent a connection request.
- **SYN-RECEIVED** - represents waiting for a confirming connection request acknowledgement after having both received and sent a connection request.
- **ESTABLISHED** - represents an open connection, data received can be delivered to the user. This is the normal state for the data transfer phase of the connection.
- **FIN-WAIT-1** - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
- **FIN-WAIT-2** - represents waiting for a connection termination request from the remote TCP.
- **CLOSE-WAIT** - represents waiting for a connection termination request from the local user.
- **CLOSING** - represents waiting for a connection termination request acknowledgment from the remote TCP.
- **LAST-ACK** - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).
- **TIME-WAIT** - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.
- **CLOSED** - represents no connection state at all.

A TCP connection progresses from one state to another in response to events. The events are: user calls, e.g. **OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATU**S; the incoming segments, particularly those containing the **SYN, ACK, RST** and **FIN** flags; and timeouts.

The state diagram in Figure 4 illustrates only state changes, together with the causing events and causing actions, but addresses neither error conditions nor actions that are not connected with state changes.

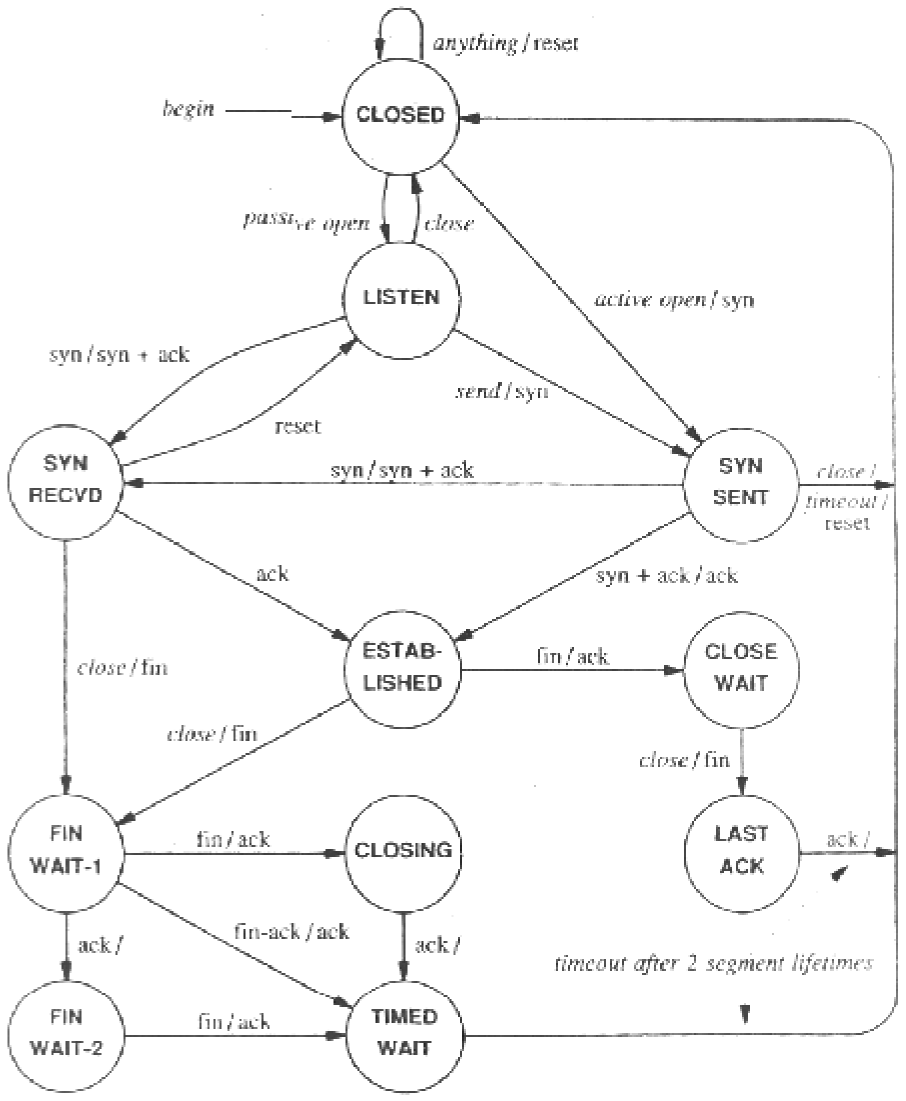

Figure 4.TCP Connection State Diagram ([2])

1.2 NuSMV

NuSMV ([5]) is a symbolic model checker jointly developed by Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST). The NuSMV project aims at the development of an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas.

The main features of NuSMV are the following:

- Functionality. NuSMV allows for the representation of synchronous and asynchronous finite state systems, and the analysis of specifications expressed in computation Tree Logic (CTL) and Linear Temporal Logic (LTL). Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried out with a textual, as well as graphical interface.

- Architecture. A software architecture has been defined. The different components and functionality of NuSMV have been isolated and separated into modules. Interfaces between modules have been provided. This should reduce the effort needed to modify and extend NuSMV.
- Quality of the implementation. NuSMV is written in ANSI C, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. NuSMV uses the state of the art BDD package developed at Colorado University. This makes it very robust, portable, efficient. Furthermore, its code should be easy to understand and modify by people other than the developers.

1.3 Well-known TCP attacks

Although TCP has been used as the reliable communication protocol for a long time, it still has some security problems, such as IP spoofing and Denial of Service attacks. Most of them are based on a well-known flaw in TCP. As Steve M.Bellovin explains in his paper [4]: "If available, the easiest mechanism to abuse is IP source routing. Assume that the target host uses the reverse of the source route provided in a TCP open request for return traffic...The attacker can then pick any IP source address desired, including that of a trusted machine on the target’s local network." ([4])

In this report, we assume that the attacker can easily forge source IP addresses. We are going to find some of the attacks using our model.

1.4 Outline of the report

The rest of the report is organized as follows: Section 2 addresses what is the model that we want to build, its inputs and outputs, and the abstractions that we make. In section 3 we verify several properties in order to prove the consistency of our model and explain why these properties are necessary for the consistency. Furthermore, we use this model to find possible ways to attack TCP. Section 4 concludes the paper presenting the summary of our work, the difficulties we met and what we learned from this project.

2. Design and implementation

In this part, we introduce the model we built; the inputs, outputs and the transitions among the states of the connections.

The model has only one process. It is built according to the TCP finite state machine. Since the ultimate goal is to find possible ways to attack TCP, we have to know how to send right segments to change the state of connection on the other side. So we focus on modeling how the arriving segments, usercalls and timeouts affect the state of connection, i.e. we model all the inputs and the state transitions, and don't model the outputs except the reset.

```
event: {USERCALL, SEGMENT, TIMEOUT};
```

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. Thus we define three kinds of inputs: Usercalls, Segments, Timeouts. The variable `event` is used to indicate which event happens.

The following is the definition of userecalls in our model:

```
usercall: {OPEN-P, OPEN-A, SEND, RECEIVE, CLOSE, ABORT};
active_flag: boolean;
```

Usercalls include OPEN, SEND, RECEIVE, CLOSE, ABORT and STATUS. We eliminate the usercall of STATUS in the model according to the TCP specification because it doesn't affect the TCP connection state at all. There are two kinds of OPEN: one is active OPEN, the other is passive OPEN, i.e. open a socket to listen. So we use OPEN-A and OPEN-P to indicate if the OPEN is active. We also have a variable `active_flag` to indicate if the latest OPEN is active. The reason we need the variable is that when the state is SYN-RECEIVED and a segment with a reset control bit arrives, the state may change according to the previous OPEN. If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then TCP should return this connection to the LISTEN state. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state), then the connection was refused. TCP should enter the CLOSED state, delete the TCB, and return. The following is the code:

```
  state   = SYN-RECEIVED :
      ...
      event = SEGMENT:
     case
       !seq_ok: SYN-RECEIVED;
       rst_flag & !active_flag: LISTEN;
       rst_flag & active_flag: CLOSED;
          ...
     esac;
```

TCP keeps trace of various segment information, including the flags of the segment and the state of the sender and the receiver. Segment variables include Send Sequence Variables, Receive Sequence Variables, Current Segment Variables (prefixed by SND, RCV and SEG respectively) and Segment Control Bit. All these variables except Segment Control Bit are used to maintain and verify the sequence numbers of segments. The typical comparisons of sequence numbers include:
(a) Determining if an acknowledgment refers to some sequence number sent but not yet acknowledged.
(b) Determining if all sequence numbers of a segment have been acknowledged (e.g., to remove the segment from a retransmission queue).

(c) Determining if an incoming segment contains sequence numbers which are expected (i.e., the segment sequence numbers overlap with those of the receive window).

For (a) and (b), we use the inequality below to judge if the acknowledgement is an "acceptable ack" :
SND.UNA < SEG.ACK =< SND.NXT
where

- SND.UNA represents the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged sequence number) and is sometimes referred to as the left edge of the send window.
- SEG.ACK acknowledges the receipt of an octet with a given sequence number, and all octets of this segment with lower sequence numbers.
- SND.NXT represents the next sequence number the local (sending) TCP will use on the connection. It is initially selected from an initial sequence number curve (ISN) and is incremented for each octet of data or sequence control transmitted.

For (c), we use the following inequality to judge if the sequence number is an "acceptable sequence number":
RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
or
RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND
where

- RCV.NXT represents the next sequence number the local TCP is expecting to receive.
- RCV.WND represents the sequence numbers the local (receiving) TCP is willing to receive.
- SEG.SEQ represents the number in the sequence field of the arriving segment.
- SEG.LEN represents the size of the packet in terms of sequence numbers.

The local TCP considers that segments in the range RCV.NXT to RCV.NXT + RCV.WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and are discarded.

Since these inequalities, rather than each individual variable, determine the state of the connection, we abstract them into two boolean variables: `seq_ok` and `ack_ok.`

SEG.PRC is the segment precedence value. In the specification of TCP, the term "security/compartment" is used to indicate the security parameters in IP including security, compartment, user group, and handling restriction. The segment precedence values of the arriving segments will be compared with the precedence value of the connection which is stored in the TCB (Transmission Control Block). The results of the comparison can only be `LOW`, `EQUAL` or `HIGH`. Any connecting attempt with a mismatched security/compartment value or an improper precedence value must be rejected. Rejecting a connection due to low a precedence value occurs only after an acknowledgment of the SYN has been received.

The following is the definition of variables related to segments:

```
seq_ok: boolean;   -- If the sequence number is acceptable
ack_ok: boolean;   -- If the ACK number is acceptable
prc_flag: {LOW, EQUAL, HIGH};   -- SEG.PRG
urg_flag: boolean;              -- URG Control Bit
ack_flag: boolean;              -- ACK Control Bit
psh_flag: boolean;              -- PSH Control Bit
rst_flag: boolean;              -- RST Control Bit
syn_flag: boolean;              -- SYN Control Bit
fin_flag: boolean;              -- FIN Control Bit
```

The variable `timeout` is defined below:

```
timeout: {USER-TIMEOUT, RETRANSMISSION-TIMEOUT,
TIMEWAIT-TIMEOUT};
```

The variable `state` is defined below:

```
state: {LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED,
FIN-WAIT-1,FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK,
TIME-WAIT, CLOSED};
```

It is recommended by [1] that we should model TCP according to the Robustness Principle: be conservative in what you do, be liberal in what you accept from others. So all the inputs, which include e`vent`, `usercall`, `timeout`, `seq_ok`, `ack_ok`, `prc_flag`, `urg_flag`, `ack_flag`, `psh_flag`, `rst_flag`, `syn_flag` and `fin_flag`, set a nondeterministically. For example, suppose that a segment arrives. The SYN control bit is assigned as below:

```
next(syn_flag) := {0, 1};
```

But the `active_flag` cannot be set in a non-deterministic way because it is related to the usercall OPEN. We define the next value of `active_flag` below:

```
next(active_flag) := case
    event = USERCALL & usercall = OPEN-A: 1;
    event = USERCALL & usercall = OPEN-P: 0;
    1: active_flag;
  esac;
```

The transitions between states are more complicated. We deal with three events separately, and only one of the three events is processed at a time. For each state, we process the segment by checking flags of the segments in a fixed sequence. For example, if the state is SYN-SENT and a segment arrives, the process is:

1. Check the ACK bit. If the ACK bit is set but is not acceptable, the segment is dropped;
2. If the RST bit is set and the ACK is acceptable, the next state is CLOSED;
3. If the RST bit is set and the ACK is not acceptable, the segment is dropped;

4. If the security/compartment in the segment does not exactly match the security/compartment in the TCB, the next state is still SYN-SENT;
5. If the SYN bit is set and the ACK is acceptable, the next state is ESTABLISHED;
6. If the SYN bit is set and the ACK is not acceptable, the next state is SYN RECEIVED;
7. Otherwise, the next state is still SYN -SENT.
The following is a part of the source code related to the procedure above:

```
    event = SEGMENT:
     case
       ack_flag & !ack_ok: SYN-SENT;
       rst_flag & ack_ok: CLOSED;
       rst_flag & !ack_ok: SYN-SENT;
       !(prc_flag = EQUAL) : SYN-SENT;
       syn_flag & ack_ok: ESTABLISHED;
       syn_flag & !ack_ok: SYN-RECEIVED;
       1: SYN-SENT;
     esac;
```

The processing of usercalls is relatively simple. For each usercall, we assign a value to the variable `state` according the current state of connection.

The processing of timeouts is also quite straightforward. Among the three timeouts, we only deal with USER TIMEOUT. If it occurs, the state of a connection will change to `CLOSED`.

The reason we put the reset into the model is because "the reset is sent" always means the arriving segment is wrong or the connection will be closed. It's useful when we want to determine what kind of segments we should send in order to get the right response. We separate the situations into 3 cases:

1. If the connection does not exist (`CLOSED`) then a reset is sent in response to any incoming segment except another reset.

```
state = CLOSED  & !rst_flag: 1;
```

2. If the connection is in any non-synchronized state (`LISTEN, SYN-SENT, SYN-RECEIVED`), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if an incoming segment has an incorrect security level/compartment, a reset is sent.

```
(state = LISTEN | state = SYN-SENT | state = SYN-RECEIVED) &
 ((ack_flag & !ack_ok) | !(prc_flag = EQUAL)): 1;
```

3. If the connection is in a synchronized state (`ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT`), for any unacceptable segment (out of window sequence number or unacceptable acknowledgment number), the connection remains in the same state. If an incoming segment has an incorrect security level/compartment, or precedence, a reset is sent.

```
(state = ESTABLISHED | state = FIN-WAIT-1 |
 state = FIN-WAIT-2 | state = CLOSE-WAIT | state = CLOSING |
 state = LAST-ACK | state = TIME-WAIT) &
 (!seq_ok | (ack_flag & !ack_ok) | !(prc_flag = EQUAL)): 1;
```

If there is an `ABORT` usercall, and the connection state is `SYN-RECEIVED` or `ESTABLISHED` or `FIN-WAIT-1` or `FIN-WAIT-2` or `CLOSE-WAIT`, a reset is also sent.

3. Verification and Exploitation

In this section, we first verify the consistency of our model, and then exploit TCP to find some possible attacks.

3.1 Consistency

Our ultimate goal is to find out the possible ways of attacking TCP, so all the verification and exploitation should be based on a consistent model.

3.1.1 Transitions among the states of the TCP state machine

Based on Figure 4, there are 22 transitions in the TCP state machine. We successfully verified 17 of them. We also discuss why not all transitions could be verified. We explain four representable properties in detail below.

Note: For each transition, we first verify the existence of left-hand side condition.

(1) From `LISTEN` to `SYN-RECEIVED:`

```
SPEC EF(state = LISTEN & event = SEGMENT & !rst_flag &
!ack_flag & syn_flag & (prc_flag = EQUAL))

SPEC AG((state = LISTEN & event = SEGMENT & !rst_flag &
!ack_flag & syn_flag & (prc_flag = EQUAL)) ->
AX(state = SYN-RECEIVED))
```

This property states that if TCP is in state `LISTEN` and a segment which contains the `SYN` control bit and doesn't contain `RST` and `ACK`, and has correct precedence/security level arrives, then the next state is `SYN-RECEIVED`.

`(2)` From `LISTEN` to `SYN-SENT:`

```
SPEC EF(state = LISTEN & event = USERCALL & usercall = SEND)
SPEC AG((state = LISTEN & event = USERCALL &
usercall = SEND) -> AX(state = SYN-SENT))
```

This property states that if TCP is in state `LISTEN` and the incoming event is a `SEND` usercall, then the next state must be `SYN-SENT`.

(3) From `SYN-SENT` to `ESTABLISHED`

```
SPEC EF(state = SYN-SENT & event = SEGMENT &
(!ack_flag | (ack_flag & ack_ok)) & !rst_flag &
(prc_flag = EQUAL) & syn_flag)

SPEC AG((state = SYN-SENT & event = SEGMENT &
(!ack_flag | (ack_flag & ack_ok)) & !rst_flag &
(prc_flag = EQUAL) & syn_flag) -> AX(state = ESTABLISHED))
```

This property states that if TCP is in the state `SYN-SENT`, and all of the following conditions hold, then it will enter state `ESTABLISHED.` The conditions are: first, the incoming segment contains the correct acknowledge number with an `ACK` control bit or doesn't contain the `ACK` control bit at all; second, it doesn't contain RST control bit. Finally, it has the right precedence/security level and `SYN` control bit.

(4) From `SYN-RECEIVED` to `ESTABLISHED`

```
SPEC EF(state = SYN-RECEIVED & event = SEGMENT & seq_ok &
!rst_flag &  (prc_flag = EQUAL) & !syn_flag & ack_flag &
ack_ok & !fin_flag)

SPEC AG((state = SYN-RECEIVED & event = SEGMENT & seq_ok &
!rst_flag &  (prc_flag = EQUAL) & !syn_flag & ack_flag &
ack_ok & !fin_flag) -> AX(state = ESTABLISHED))
```

This property states that in the state `SYN-RECEIVED`, if all of the following conditions hold, TCP will enter the state `ESTABLISHED`. First, the incoming segment has a correct sequence number. Second, the incoming segment does not contain RST or SYN or FIN control bit. Third, the precedence/security level of the segment is right. Finally, the incoming segment contains an ACK with the correct acknowledge number.

The reason why we give these four transitions in the diagram as an example is because there are only 2 states (`SYN-SENT` and `SYN-RECEIVED`) that may lead to the state of `ESTABLISHED`. So these 2 states are most likely to be the target of the attack; and the `LISTEN` state is at the very beginning of establishing a condition in TCP.

The other 14 properties are also proved to be true. We verified them on a Sun Ultra-Enterprise 3000 machine with 1G memory using SunOS 5.51 operating system. The following are some of the running statistics for the verification:

```
------------------
User time    1.697 seconds
System time  0.295 seconds
Virtual text size              =   770K
Virtual data size              = 54826K
    data size initialized      =    87K
    data size uninitialized      =  1107K
    data size sbrk               = 53632K
Virtual memory limit             = 2097148K (2097152K)

BDD statistics
--------------------
BDD nodes allocated: 3743
Monolithic Transition Relation:
BDD nodes representing transition relation: 266 + 1
```

The following are the five transitions that we could not verify in our model.

```
(1) FIN-WAIT-1 --> CLOSING
(2) FIN-WAIT-1 --> FIN-WAIT-2
(3) FIN-WAIT-1 --> TIME-WAIT
(4) CLOSING --> TIME-WAIT
(5) LAST-ACK --> CLOSED
```

The reason is that all the transitions from the left-hand side to the right-hand side need the information of whether the coming ACK acknowledgment is in response to the FIN message that we have sent before. But in our model, we don't store the information that we sent out, and we don't know if the ACK is in response to our FIN or not. The model does not differentiate between these two conditions, non-deterministically transitioning into `FIN-WAIT-1` or `FIN-WAIT-2`. For example,

```
event = SEGMENT:
      case
        !seq_ok: FIN-WAIT-1;
        rst_flag: CLOSED;
        syn_flag: CLOSED;
        !ack_flag: FIN-WAIT-1;
        !ack_ok: FIN-WAIT-1;
        ack_ok & !fin_flag: {FIN-WAIT-1, FIN-WAIT-2};
-- If our FIN is now acknowledged then enter FIN-WAIT-2 and
-- continue processing in the state of FIN-WAIT-1.
        fin_flag: {TIME-WAIT, CLOSING};
-- If our FIN has been ACKed then enter TIME-WAIT,
-- otherwise enter the CLOSING state.
        1: FIN-WAIT-1 ;
      esac;
```

3.1.2 Other TCP behaviors

We also verified some properties related to resets. They are useful in detecting wrong incoming segments. As mentioned in Section 2, these transitions can be separated into 3 cases.

1 If the connection does not exist (CLOSED) then a reset is sent in response to any incoming segment except another reset.

```
SPEC EF(state = CLOSED  & ack_flag & !rst_flag &
event = SEGMENT)

SPEC AG((state = CLOSED  & ack_flag & !rst_flag &
event = SEGMENT) -> (out_rst))
```

2 If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if the incoming segment has an improper security level or compartment, a reset is sent. The following is an example of state SYN-RECEIVED:

```
SPEC  EF(state  =  SYN-RECEIVED  &  seq_ok  &  !rst_flag  &
!syn_flag & ack_flag & !ack_ok & event = SEGMENT)

SPEC  AG((state  =  SYN-RECEIVED  &  seq_ok  &  !rst_flag  &
!syn_flag & ack_flag & !ack_ok & event = SEGMENT)
-> (AX(state = SYN-RECEIVED) & AF(out_rst)))
```

We have to set `!rst_flag & !syn_flag` to guarantee that the incoming acknowledgment is an ACK when the state is SYN-RECEIVED. The other two states are very similar. Please refer to the appendix.

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), for any unacceptable segment (out of window sequence number or unacceptable acknowledgment number), the connection remains in the same state. If an incoming segment has an incorrect security level/compartment, or precedence, a reset is sent

```
SPEC  EF(state  =  ESTABLISHED  &  !rst_flag  &  !syn_flag  &
ack_flag & (!ack_ok | !seq_ok) & event = SEGMENT)

SPEC  AG((state  =  ESTABLISHED  &  !rst_flag  &  !syn_flag  &
ack_flag & (!ack_ok | !seq_ok) & event = SEGMENT)
-> (AX(state = ESTABLISHED) & AF(out_rst)))
```

All the properties are verified to be true. The following are some of the runtime statistics

```
------------------
User time    2.844 seconds
System time  0.288 seconds
Virtual text size                =   770K
Virtual data size                = 54978K
    data size initialized        =    87K
    data size uninitialized      =  1107K
    data size sbrk               = 53784K
Virtual memory limit             = 2097148K (2097152K)

BDD statistics
--------------------
BDD nodes allocated: 4465
Monolithic Transition Relation:
```

```
BDD nodes representing transition relation: 274 + 1
```

3.2 Exploitation

As we mentioned in Section 1, one kind of possible attacks on TCP is denial-of-service (DoS), such as TCP SYN flooding[3]. Most DoS attacks do not exploit a software bug, but rather a shortcoming in the particular implementation of a protocol. We first use our model to confirm the TCP SYN flooding attack and then find a new way to attack TCP.

In [3], we know that every time a client SYN arrives on a valid port (a port where a TCP server is listening), a TCB must be allocated. If there were no limit on the number of concurrent connection requests, a busy host can easily exhaust all of its memory just trying to process TCP connections. However, TCP has an upper limit to the amount of concurrent connection requests, which is called backlog. It is the length of the queue where incoming (and yet incomplete) connections are kept. This queue limit applies to both the number of incomplete connections (the 3-way handshakes that have not been completed) and the number of completed connections that have not been pulled from the queue. If this backlog limit is reached, TCP silently discards all incoming connection requests until the pending connection requests can be dealt with.

First, we want to use NuSNV to produce a counter example to show the existence of TCP SYN flooding attacks, and then we will construct an attack and verify whether it will always be successful.

Because TCP SYN flooding attacks happen when the state of connections is `SYN-RECEIVED`, we want to verify that if the current state is `SYN-RECEIVED` then finally the state will be `ESTABLISHED.` The following is the property:

```
AG (state = SYN-RECEIVED -> AF state = ESTABLISHED)
```

Also because there are no usercalls or incoming segments for a given socket when TCP SYN flooding attacks happen, we add fairness as follows:

```
FAIRNESS
state = SYN-RECEIVED ->
!(event = USERCALL | event = SEGMENT)
```

NuSMV produced a counter example (Appendix B). We use Figure 5 below to illustrate the example. This counter example states that if `USER TIMEOUT`s happen infinitely often when the state is `SYN-RECEIVED` then the state will not be `ESTABLISHED`, which shows the possibility of TCP SYN flooding attacks.

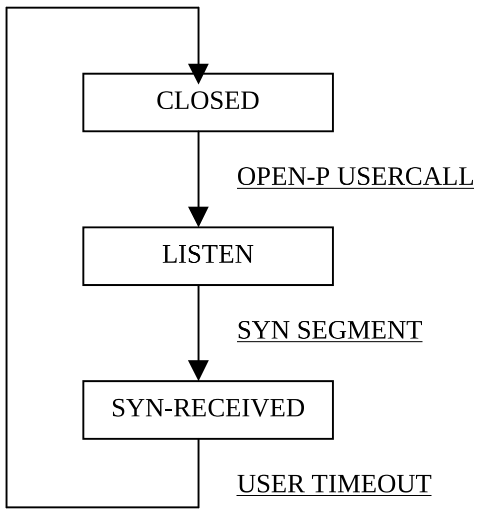

Figure 5. The counter example produced by NuSMV

We now set out to find some of these attacks. First, we want to construct a SYN segment that can be accepted by a listening target host. There are several important parts in the TCP head of a segment: IP address, Sequence number and Control bits. According to the TCP specification, we know that the other side doesn't verify the sequence number of the SYN segment. So it may have any sequence number. We can use the real IP address in the segment. However, it's not a good idea. The best way to attack is to forge an IP address so that nobody can find where the segment comes from. But the forged IP address should be unreachable; otherwise, a reset from the IP address will be sent. Here is the property:

```
SPEC AG(event = SEGMENT & state = CLOSED & !rst_flag
-> out_rst)
```

This property means that if a segment without RST arrives and the current state of the socket in the forged IP address is `CLOSED`, then a reset will be sent to the target host. If the target receives the reset, it may close the connection. In this case, we will not fill up the backlog. The property below expresses this idea:

```
SPEC AG(event = SEGMENT & (state = SYN-SENT | state = SYN-
RECEIVED | state = ESTABLISHED) & rst_flag & ack_flag &
ack_ok -> AX state = CLOSED)
```

Now we will construct the segment. We start by checking whether it is possible that the state will change from `LISTEN` to `SYN-RECEIVED` when a SYN segment arrives. Here is the property:

```
SPEC EF(event = SEGMENT & state = LISTEN & syn_flag ->
AX(state = SYN-RECEIVED))
```

The property is verified to be true. Then we want to check whether this attack will always be successful. If we simply replace "`EF`" with "`AG`" in the property above, it doesn't hold any longer. The reason is that there are some other control bits we should set or reset, such as `RST`, `ACK` and `SEG.PRC`. The following is the correct property:

```
SPEC AG(event = SEGMENT & state = LISTEN & syn_flag &
!rst_flag & !ack_flag & prc_flag = EQUAL ->
AX(state = SYN-RECEIVED))
```

From all the properties we have verified, we conclude that there are three steps in this attack:

1. Send a correct SYN segment to the victim host;
2. Do not response to any messages from the victim host and let the connection timeout. One way of doing this it to forge an unreachable IP address in the SYN segment above.
3. When the available memory in the backlog of the victim host is used up, all the connection requests to the host will be discarded, i.e., the host is under TCP SYN flooding attack.

We use the following example to illustrate the attack.

```
1    Z(x)  --- SYN --->        A
     Z(x)  --- SYN --->        A
     Z(x)  --- SYN --->        A
     Z(x)  --- SYN --->        A
           ...
2    X  <--- SYN/ACK ---    A
     X  <--- SYN/ACK ---    A
           ...
3    Y      --- SYN --->        A (drop the SYN segment)
           ...
```

In step 1 the attacking host Z sends a multitude of SYN requests with a forged IP address X to the target A to fill its backlog queue with pending connections. In step 2, the target responds with SYN/ACKs to what it believes is the source of the incoming SYNs. In step 3, when the backlog queue is filled up, all further requests to this TCP port will be ignored. The target port is flooded.

Using this model, we found another way to attack target hosts through denial-of-service. We noticed that the arriving segments with SYN control bit may cause the connection to close. So we wondered if the connection will close when there is a segment with the correct sequence number and SYN control bit when the current state is neither `LISTEN` nor `SYN-SENT`. We checked the following property:

```
SPEC AG(!(state = LISTEN | state = SYN-SENT) &
event = SEGMENT & seq_ok & syn_flag ->   AX(state = CLOSED))
```

However, this property does not hold. The reason is that TCP protocol checks RST control bit and Security and Precedence flag before SYN flag. So we needed to set the right RST and SEQ.PRC flag.

```
SPEC AG(!(state = LISTEN | state = SYN-SENT) &
event = SEGMENT & seq_ok & !rst_flag & (prc_flag = EQUAL) &
syn_flag -> AX(state = CLOSED))
```

The property above was verified to be true. Thus, the following attack is realistic:
- The cracker knows that there is a connection between the hosts A and B
- He wants to disrupt the connection.
- To do so, he must let either A or B close the connection.
- Suppose he want to close the connection at the side of A. He forges the address of B and sends the SYN segment to A. Then A will close the connection.

So if the cracker can detect and disrupt every connection that the target host wants to establish, the target host is under a denial of service attack.

The difficulty of this attack is in getting the right sequence number. The two ways of obtaining it are guessing, catching a possibility [4], and sniffing. If the adversary can sniff the network, he can disrupt all connections except his. Sniffing is hard to detect, especially if the adversary is using a system of network cards instead of TCP.

4. Conclusion

In this report, we not only modeled the TCP successfully, but also confirmed a well-known TCP SYN flooding attack ([3]) and found a new way to attack TCP.

Since a good understanding of TCP is a prerequisite, much time was spent on reading relevant papers and documentation. The most difficult part was perhaps finding the right tool (SPIN or NuSMV) and modeling at the necessary level of abstraction.

The result of this project has demonstrated that formal methods are usable in verifying protocols. They also can be used to model complex systems. However, a reasonable abstraction of the system may be necessary.

5. References

1. Marina del Rey, Transmission Control Protocol Darpa Internetprogram Protocol Specification. California 1981

2. Douglas E. Comer, Internetworking with TCP/IP Vol 1: Principles, protocols, and Architecture (Third Edition). Prentice-Hall, May 1995

3. Project Neptune, Phrack Magazine, Volume Seven, Issue forty-eight, File 13 of 18, July 1996

4. S.M Bellovin, Security Problems in the TCP/IP Protocol Suite, Computer Communication Review, Vol. 19, No.2 pp.32-48, April 1989

5. Alessandro Cimatti and Marco Roveri, NuSMV 1.1 User Manual