Misbehaviors in TCP SACK Generation

Nasif Ekiz
nekiz@udel.edu

Abuthahir Habeeb Rahman
abu@udel.edu

Paul D. Amer
amer@udel.edu

Computer and Information Sciences Department
University of Delaware
Newark, Delaware 19716

ABSTRACT

While analyzing CAIDA Internet traces of TCP traffic to detect instances of data reneging, we frequently observed seven misbehaviors in the generation of SACKs. These misbehaviors could result in a data sender mistakenly thinking data reneging occurred. With one misbehavior, the worst case could result in a data sender receiving a SACK for data that was transmitted but never received. This paper presents a methodology and its application to test a wide range of operating systems using TBIT to fingerprint which ones misbehave in each of the seven ways. Measuring the performance loss due to these misbehaviors is outside the scope of this study; the goal is to document the misbehaviors so they may be corrected. One can conclude that the handling of SACKs while simple in concept is complex to implement.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol Verification; C.2.5 [**Local and Wide-Area Networks**]: Internet – TCP

General Terms

Reliability, Verification

Keywords

SACK, Selective Acknowledgment, TBIT, TCP

1. INTRODUCTION

The Selective Acknowledgment (SACK) mechanism, RFC2018 [11], an extension to Transmission Control Protocol's (TCP) [15] ACK mechanism, allows a data receiver to explicitly acknowledge arrived *out-of-order* data to a data sender. When using SACKs, a TCP data sender need not retransmit SACKed data during the loss recovery period. Previous research [1, 5, 8] showed that SACKs improve TCP throughput when multiple losses occur within the same window. The success of SACK-based loss recovery algorithm [3] is proportional to the SACK information received from the data receiver. In this paper, we investigate RFC2018 conformant SACK generation.

Deployment of the SACK option in TCP connections has been a slow, but steadily increasing trend. In 2001, 41% of the web servers tested were SACK-enabled [13]. In 2004, SACK-enabled web servers increased to 68% [12]. All of the operating systems tested in this study accept SACK-permitted TCP connections.

Today's reliable transport protocols such as TCP and Stream Control Transmission Protocol (SCTP) [16] are designed to tolerate data receiver reneging (simply, data reneging) (Section 8 RFC2018). Data reneging occurs when a data receiver SACKs data, and later discards that data from its receiver buffer prior to delivering it to a receiving application (or receiving socket buffer).

In related research, we argue that reliable transport protocols should not be designed to tolerate data reneging; largely because we believe data reneging rarely if ever occurs in practice [7]. While developing our software to discover data reneging in trace data, we analyzed TCP SACK information within Internet traces provided by the Cooperative Association for Internet Data Analysis (CAIDA) [6]. At first it seemed that data reneging was happening frequently. On closer inspection however, it appears that the generation of SACKs in many TCP connections potentially was incorrect according to RFC2018. Sometimes SACK information that should have been sent was not. Sometimes the wrong SACK information was sent. In one misbehavior, SACKs from one connection are sent in the SYN-ACK used to open a later connection! These misbehaviors wrongly gave the impression that data reneging was occurring.

Our discovery led us to verifying SACK generation behavior of TCP data receivers for a wide range of operating systems. In this paper, our goal is to present a methodology for verifying SACK behavior, and to apply the methodology to report misbehaving TCP stacks. The goal of the paper is not to measure how much the misbehaviors degrade the performance, but rather to identify misbehaving TCP stacks so they will be corrected.

We first present in Section 2 seven misbehaviors, five (A-E) observed in the CAIDA traces, and two (F-G) additional SACK related misbehaviors observed during our testing of A-E. Technically, misbehaviors A-E indicate that SHOULD requirements of RFC2018 are not being followed, and SHOULD means "that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course." Upon analysis, we believe these misbehaviors to be accidental, not incidental.

Misbehaviors A-F can reduce the effectiveness of SACKs. Misbehavior G is the worst one where a data receiver transmits a SACK for data that was never received, thus questioning the data transfer reliability of the connection. To discover which implementations are misbehaving, we defined seven test extensions to the TCP Behavior Inference Tool (TBIT) [19], a tool that verifies TCP endpoint behavior.

The methodology using TBIT is described in Section 3, and the results of our TBIT tests are presented in Section 4. Section 5 identifies related research used to infer TCP behavior, and Section 6 concludes our work.

2. TESTING SACK BEHAVIOR

The five SACK generation misbehaviors observed in CAIDA traces are described as:

A. Fewer than max number of reported SACKs
B. Receiving data between CumACK and first SACK
C. Receiving data between two previous SACKs
D. Failure to report SACKs in FIN segments
E. Failure to report SACKs during bidirectional data flow

The two additional SACK-related misbehaviors observed during our TBIT testing of A-E are:

F. Mishandling of data due to SACK processing
G. SACK reappearance in consecutive connections

A. Fewer than Max Number of Reported SACKs

RFC2018 Section 3 specifies that "*the data receiver SHOULD include as many distinct SACK blocks possible in the SACK option,*" and that "*the 40 bytes available for TCP options can specify a maximum of four SACK blocks*." For some TCP flows, we observed that only two or sometimes three SACK blocks were reported by a data receiver even though additional space existed in the TCP header.

That is, more than two SACK blocks at the data receiver are known to exist (say X_l-X_r, Y_l-Y_r, and Z_l-Z_r) but only two SACK blocks are reported (X_l-X_r and Y_l-Y_r). When the cumulative ACK advances beyond X_r, SACK block X_l-X_r, is correctly no longer reported, and SACK block Z_l-Z_r is reported along with block Y_l-Y_r. This misbehavior implies that the data receiver reports less than the recommended maximum SACK blocks.

We extended the existing TBIT test "SackRcvr" [19] to determine a receiver's maximum number of reported SACK blocks. For clarity, most TCP segments sent by TBIT in our Figs. 1-7 are shown to carry 1 byte of data and create 1 byte gaps. This numbering scheme makes the TBIT tests easy to understand. In the actual tests performed (see traces [17]), segments carry 1460 bytes of data and create 1460 byte gaps. The only exception was for Tests A,F for Linux systems. The Linux advertised receiver window is only 5840 bytes. To simulate 4 gaps, TBIT segments for two Linux tests carry 600 bytes of data and create 600 byte gaps.

The TBIT test in Fig. 1 operates as follows. Sequence numbers of segments are shown in parenthesis:

Test A

1. TBIT establishes a connection to TCP Implementation Under Test (IUT) with SACK-Permitted option and Initial Sequence Number (ISN) 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (405) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK
9. TBIT sends segment (407) creating 3rd gap at IUT
10. IUT acks the out-of-order data with SACK
11. TBIT sends segment (409) creating 4th gap at IUT
12. IUT acks the out-of-order data with SACK
13. TBIT sends three resets (RST) to abort the connection

The last SACK from the IUT reflects an implementation's support for maximum number of SACK blocks reported. A conformant implementation's last SACK should be as SACK #12 in Fig. 1. A misbehaving implementation would not SACK block Y (Misbehavior A1), or blocks X and Y (Misbehavior A2).

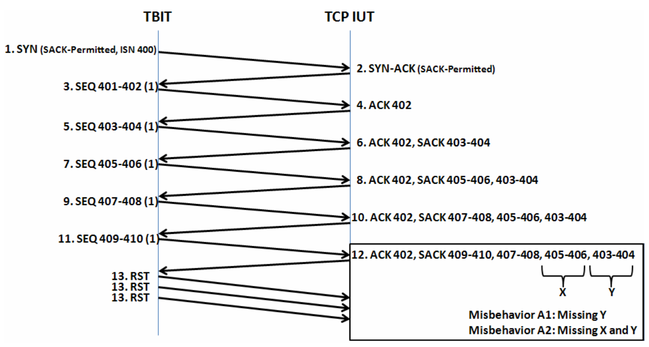
TBIT
TCP IUT
1. SYN (SACK-Permitted, ISN 400)
2. SYN-ACK (SACK-Permitted)
3. SEQ 401-402 (1)
4. ACK 402
5. SEQ 403-404 (1)
6. ACK 402, SACK 403-404
7. SEQ 405-406 (1)
8. ACK 402, SACK 405-406, 403-404
9. SEQ 407-408 (1)
10. ACK 402, SACK 407-408, 405-406, 403-404
11. SEQ 409-410 (1)
12. ACK 402, SACK 409-410, 407-408, 405-406, 403-404
X
Y
13. RST
13. RST
13. RST
Misbehavior A1: Missing Y
Misbehavior A2: Missing X and Y

Figure 1: Fewer than max number of reported SACKs

B. Receiving Data Between CumACK and First SACK

For some TCP flows having at least two SACK blocks, we observed the following misbehavior. Once the data between the cumulative ACK and the first SACK block was received, the data receiver increased the cumulative ACK, but misbehaved and did not acknowledge other SACK blocks. (The acknowledgment with no SACK blocks implies an instance of data reneging.)

RFC2018 specifies that: "*If sent at all, SACK options SHOULD be included in all ACKs which do not ACK the highest sequence number in the data receiver's queue.*" So, SACKs should be included when the cumulative ACK is increased and out-of-order data exists in the receive buffer.

Test B, illustrated in Fig. 2, checks this misbehavior. The second SACK block should remain present when the cumulative ACK is increased beyond the first SACK block but is less than the second SACK block.

Test B

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (404) creating a gap at IUT (the gap between Cum ACK and first SACK block)
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (406) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK
9. TBIT sends segment (403)
10. IUT acks the out-of-order data with SACK
11. TBIT sends segment (402) to fill the gap between Cum ACK and first SACK
12. IUT acks the in order data with SACK
13. TBIT sends three RSTs to abort the connection

A conformant implementation should report SACK block (406-407) as shown in #12 in Fig. 2. A misbehaving implementation omits reporting the SACK block.

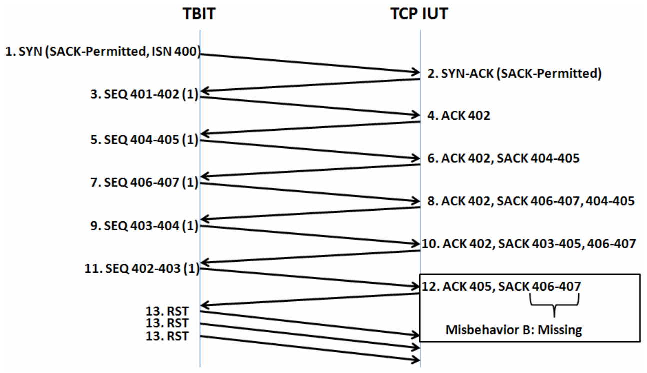

Figure 2: Receiving data between CumACK and first SACK

C. Receiving Data Between Two Previous SACKs

We observed that some TCP flows report SACK information incompletely once the missing data between two SACK blocks (say X_l-X_r and Y_l-Y_r) are received. The next SACK should report a single SACK block concatenating the first SACK block (X_l-X_r), the missing data in between, and the second SACK block (Y_l-Y_r). Instead some implementations generate a SACK covering only the first SACK block and the missing data, i.e., (X_l-Y_l), omitting the second SACK block. This behavior implies that the second SACK block is reneged.

Test C, illustrated in Fig. 3, tests this misbehavior. The data receiver should report one SACK block covering the two SACK blocks and the data in between.

Test C

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (405) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK
9. TBIT sends segment (404) with missing data between the first and the second SACK blocks
10. IUT acks the out-of-order data with SACK
11. TBIT sends three RSTs to abort the connection

A proper implementation is expected to report the out-of-order data (403-406) as shown in #10 in Fig. 3. A misbehaving implementation would report the SACK block partially (403-405).

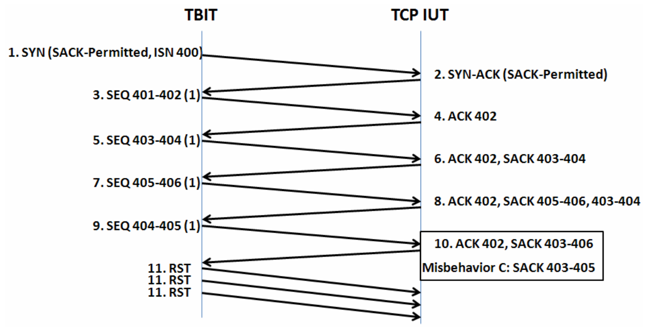
TBIT
TCP IUT
1. SYN (SACK-Permitted, ISN 400)
2. SYN-ACK (SACK-Permitted)
3. SEQ 401-402 (1)
4. ACK 402
5. SEQ 403-404 (1)
6. ACK 402, SACK 403-404
7. SEQ 405-406 (1)
8. ACK 402, SACK 405-406, 403-404
9. SEQ 404-405 (1)
10. ACK 402, SACK 403-406
Misbehavior C: SACK 403-405
11. RST
11. RST
11. RST

Figure 3: Receiving data between two previous SACKs

D. Failure to Report SACKs in FIN Segments

When closing a connection, a receiving side sends a FIN segment along with the acknowledgment (ACK and SACK) for the data received. But for some data flows, we observed the FIN segment does not carry SACK information. As discussed in *Section 2B*, the receiver should include the SACK information along with the ACK.

Test D, in Fig. 4, operates as follows: TBIT opens a connection and sends a GET request (HTTP/1.0) to the IUT. The IUT sends the requested data, and immediately closes the connection with a FIN since HTTP/1.0 is non-persistent.

Test D

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401-450: GET /index.pdf HTTP/1.0 request) in order
4. IUT acks the in order data with ACK (450)
5. IUT starts sending segments with contents of index.pdf
6. TBIT sends segment (451) creating a gap at IUT
7. TBIT acks segments of IUT
8. IUT acks the out-of-order data with SACK
9. IUT continues sending contents of index.pdf with SACK
10. Once index.pdf is sent completely, IUT sends a FIN to close the connection

The conformed behavior of a data receiver is to include SACK information in the FIN segment as shown in #10 in Fig. 4. A misbehaving implementation sends an ACK, but no SACK information.

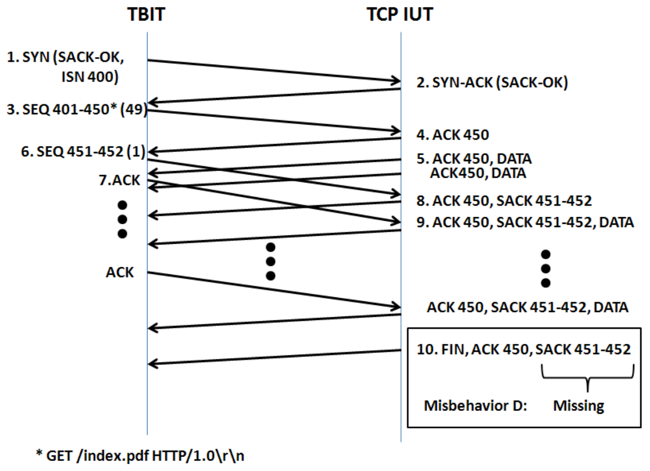

Figure 4: Failure to report SACKs in FIN segments

E. Failure to Report SACKs During Bidirectional Data Flow

This misbehavior occurs when the data flow is bidirectional. In some TCP flows, SACK information is not conveyed when the TCP segment carries data. If a TCP host is sending data continuously (e.g., an HTTP server), only one SACK is sent when out-of-order data are received, and SACK information is not piggybacked with the following segments. This misbehavior can cause less efficient SACK-based loss recovery since SACKs are sent only once for each out-of-order data arrival.

As stated in *Section 2B*, a conformant data receiver should include SACK information with all ACKs. If ACKs are piggybacked while sending data, SACKs should also be piggybacked in the TCP segments.

We added a new TBIT test for misbehavior E. To have bidirectional data flow and out-of-order data simultaneously, we used HTTP/1.1 GET requests [9]. HTTP/1.1 opens a persistent connection between TBIT and an IUT. TBIT requests the file index.pdf (11650 bytes) which is large enough to have a data transfer requiring several round trips so that SACK information can be observed in the segments.

Test E

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401-450: GET /index.pdf HTTP/1.1 request) in order
4. IUT acks the in order data with ACK (450)
5. IUT starts sending segments with contents of index.pdf
6. TBIT sends segment (451) creating a gap at IUT
7. TBIT acks segments of IUT
8. IUT acks the out-of-order data with SACK
9. IUT continues sending contents of index.pdf with SACK
10. Once index.pdf is retrieved completely, TBIT sends three RSTs to abort the persistent connection

A conformant implementation appends SACK information in TCP segments carrying data as shown in Fig. 5, whereas a misbehaving implementation does not.

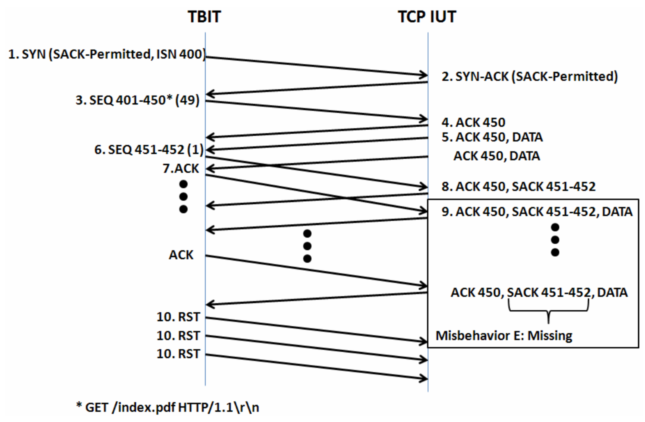

Figure 5: Failure to report SACKs during bidirectional data

F. Mishandling of Data Due to SACK Processing

While running Test E, we observed another SACK related misbehavior. Some segments do not carry maximal payload when SACKs are included. Rather they carry only the number of bytes equal to the SACK information appended.

We explain the misbehavior in detail using Test F shown in Fig. 6. Test F modifies Test E. Instead of sending one out-of-order data, four are sent to check how data is sent by the TCP IUT as the number of appended SACKs increases.

Test F

1-5. Same as Test E

6. TBIT sends segment (451) creating a gap at IUT, and ACKing the 1st segment of IUT
7. When the ACK for 1st segment of IUT is received, IUT's congestion window (cwnd) is increased enabling sending two new segments. IUT sends two segments with one SACK block: 3rd segment (1448 bytes) and 4th segment (12 bytes)
8. TBIT sends segment (453) creating a second gap at IUT, and ACKing the 2nd segment of IUT
9. When the ACK for 2nd segment of IUT is received, IUT sends two segments each with two SACKs: 5th segment (1440 bytes) and 6th segment (20 bytes)
10. TBIT sends segment (455) creating a third gap at IUT, and ACKing the 3rd segment of IUT
11. When the ACK for 3rd segment of IUT is received, IUT sends two segments each with three SACKs: 7th segment (1432 bytes) and 8th segment (28 bytes)
12. TBIT sends segment (457) creating a fourth gap at IUT, and ACKing the 4th segment of IUT
13. When the ACK for 4th segment of IUT is received, IUT sends two segments each with four SACKs: 9th segment (1424 bytes) and 10th segment (36 bytes)

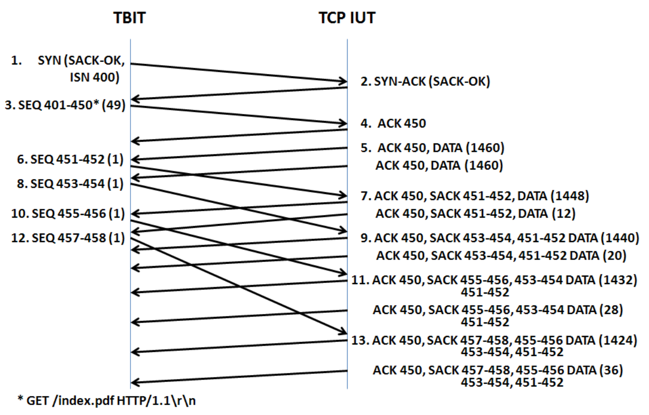
TBIT
TCP IUT
1. SYN (SACK-OK, ISN 400)
2. SYN-ACK (SACK-OK)
3. SEQ 401-450* (49)
4. ACK 450
5. ACK 450, DATA (1460)
ACK 450, DATA (1460)
6. SEQ 451-452 (1)
8. SEQ 453-454 (1)
7. ACK 450, SACK 451-452, DATA (1448)
ACK 450, SACK 451-452, DATA (12)
10. SEQ 455-456 (1)
12. SEQ 457-458 (1)
9. ACK 450, SACK 453-454, 451-452 DATA (1440)
ACK 450, SACK 453-454, 451-452 DATA (20)
11. ACK 450, SACK 455-456, 453-454 DATA (1432) 451-452
ACK 450, SACK 455-456, 453-454 DATA (28) 451-452
13. ACK 450, SACK 457-458, 455-456 DATA (1424) 453-454, 451-452
ACK 450, SACK 457-458, 455-456 DATA (36) 453-454, 451-452
* GET /index.pdf HTTP/1.1\r\n

Figure 6: Mishandling of data due to SACK processing

For every ACK received from TBIT, the IUT's cwnd is increased to send two new segments. After the first ACK is received, the IUT sends segments with 1448 and 12(!) bytes of data, respectively. Both segments from the IUT do include a SACK block. A proper SACK implementation is expected to send 1448 bytes of data in both segments each with 12 bytes of SACK in the TCP options. As the number of SACKs increase to 2, 3 and 4, the IUT sends two segments with (1440, 20), (1432, 28), (1424, 36) bytes, respectively. Note that the second segment always (coincidentally?) carries a number of data bytes equal to bytes needed for the SACK blocks, not a full size segment. This misbehavior is observed continuously while out-of-order data exists at the IUT. Throughput is decreased almost in half for the time when out-of-order data exists in the receive buffer.

G. SACK Reappearance in Consecutive Connections

When verifying misbehaviors A-E, we ran the TBIT tests successively using different port numbers. We observed that in some TCP stacks, SACK information of a prior connection, say from Test A, would sometimes appear in the SYN-ACK segment of a new connection, say from Test B!

To further investigate the misbehavior, we developed Test G as shown in Fig. 7. This test purposely uses the same initial sequence numbers for consecutive connections to demonstrate a worst case:

Test G

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400 on ephemeral port Eph_1
2. IUT replies with SACK-Permitted option on port 80
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends three RSTs segments to abort the connection
8. After 'X' minutes, TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400 on ephemeral port Eph_2
9. IUT replies with SACK-Permitted option on port 80 *including a SACK block of the previous connection*

In the second connection, the IUT sends an acknowledgment with SACK block 403-404 which is from the first connection. TBIT assumes 403 is SACKed, but the IUT never received the data. TBIT later sends data 402-403 to check if the IUT increases ACK to 405. The IUT returns an inconsistent ACK 403, SACK 403-405, but fortunately does not increase ACK to 405 so the connection remains reliable. In a real connection, eventually the sender will timeout on 403, discard all SACKed information, and retransmit the data, thus returning to a correct state [11]. However for a brief period of time, the data sender and receiver are in an inconsistent state.

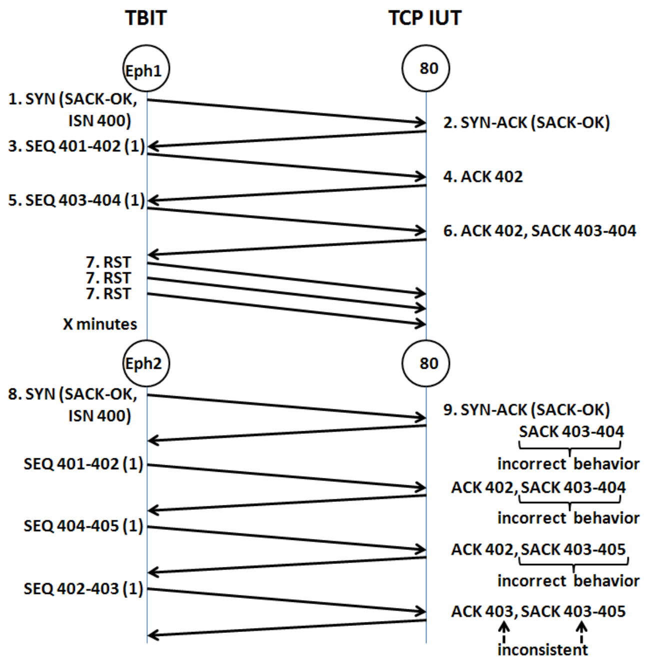

Figure 7: SACK reappearance in consecutive connections

3. EXPERIMENTAL DESIGN

The TBIT tests described in Section 2 were performed over a dedicated local area network with no loss. Tests were performed between two machines, A and B, as shown in Fig. 8. The round trip time was on average 10ms, and no background traffic was present.

The IUTs being verified were the standard TCP stacks of various operating systems. We installed 27 operating systems using Oracle's VirtualBox virtualization software [20] on machine B. We ran tests for Mac OS X on another machine.

TBIT 1.0 [19] was extended on FreeBSD 7.1 (machine A) with the seven TBIT tests detailed in the Section 2.

For each operating system, we installed an Apache HTTP Server [2] on machine B since TBIT is originally designed to infer TCP behavior of a web server. The TCP segments transmitted between TBIT and each IUT were captured at machine B. For this purpose, we also installed wireshark [21] on each Windows OS, and tcpdump [18] on each UNIX or UNIX-like OS.

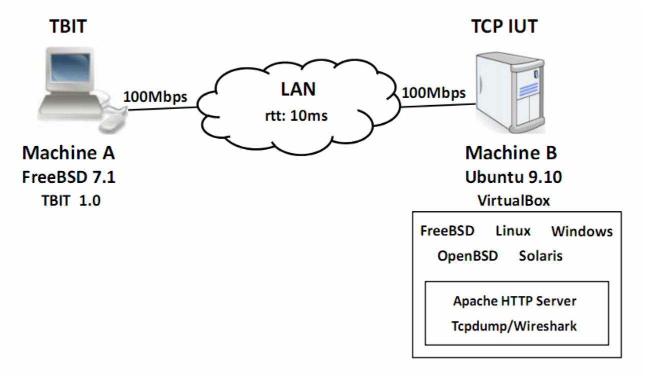

Figure 8: Experimental design

4. RESULTS

We verified the operating systems in Table I. Each TBIT test was repeated three times. In every case, all seven test outputs were consistent. Segment captures of tests and TBIT tests are available [17].

For test A, the early versions of FreeBSD, 5.3 and 5.4, and all versions of OpenBSD report at most three SACK blocks (Misbehavior A1). OpenBSD explicitly defines a parameter TCP_MAX_SACK = 3. Windows 2000, XP and Server 2003 report at most two SACK blocks (Misbehavior A2). Later Windows versions correct this misbehavior.

If the return path carrying SACKs were lossless, a TCP data receiver reporting at most two or three SACK blocks would not cause a problem. A data sender would always infer the proper state of the receive buffer for efficient SACK-based loss recovery described in RFC3517 [3]. When more than four SACK blocks exist at a data receiver, and SACK segments are lost, the chance of a data sender getting less accurate state of the receive buffer increases as SACK implementations' number of blocks reported is decreased. This misbehavior can lead to less efficient SACK-based loss recovery, and therefore decreased throughput (longer transfer times) when multiple TCP segments are lost within the same window.

We report, for test B, that Windows 2000, XP and Server 2003, are misbehaving. SACK information is not reported where it should be, after the cumulative ACK is increased beyond the first SACK block. Later Windows versions correct this misbehavior.

Misbehavior C is observed with Windows 2000, XP and Server 2003. SACK information is partially reported when the data between two previously reported SACK blocks are received. Later Windows versions correct this misbehavior.

We observed misbehavior D, failure to report SACK information in FIN segment, in FreeBSD 5.3, FreeBSD 5.4, all versions of OpenBSD and Microsoft's Windows. The problem has been corrected in the later FreeBSD versions.

Misbehavior E is observed with all versions of Windows OS. When the TCP traffic is bidirectional, SACKs are not carried within the opposite direction TCP segments. Out-of-order data are SACKed only once when they arrive. If a SACK is lost on the return path, subsequent segments with no SACKs will trigger a fast retransmission which can cause the data sender to unnecessarily retransmit data that exists in the receiver's buffer.

Table I TBIT Test Results

Operating System	**Test**							
	A1	***A2***	***B***	***C***	***D***	***E***	***F***	***G***
FreeBSD 5.3	X				X			
FreeBSD 5.4	X				X			
FreeBSD 6.0								
FreeBSD 7.3								
FreeBSD 8. 0								
Linux 2.2.20 (Debian 3)							X	
Linux 2.4.18 (Red Hat 8)							X	
Linux 2.4.22 (Fedora 1)							X	
Linux 2.6.12 (Ubuntu 5.10)							X	
Linux 2.6.15 (Ubuntu 6.06)							X	
Linux 2.6.18 (Debian 4)							X	
Linux 2.6.31 (Ubuntu 9.10)								
Mac OS X 10.5								
Mac OS X 10.6								
OpenBSD 4.2	X				X			
OpenBSD 4.5	X				X			
OpenBSD 4.6	X				X			
OpenBSD 4.7	X				X			
OpenBSD 4.8	X				X			
OpenSolaris 2008.05							X	X
OpenSolaris 2009.06							X	X
Solaris 10								X
Solaris 11							X	
Windows 2000		X	X	X	X	X		
Windows XP		X	X	X	X	X		
Windows Server 2003		X	X	X	X	X		
Windows Vista					X	X		
Windows Server 2008					X	X		
Windows 7					X	X		

The traffic pattern for testing Misbehavior E is a typical web browsing scenario. TBIT represents a user's web browser where HTTP 1.1 GET requests are pipelined, and the IUT represents an HTTP 1.1 web server. Since the scenario represents typical Internet traffic, we believe that the SACK generation misbehavior of the Windows OS is significant, and should be fixed.

Misbehavior F is observed in Solaris 11, OpenSolaris and all Linux systems except the latest one tested Linux 2.6.31 (Ubuntu 9.10), so the problem may be fixed for Linux. Interestingly, misbehavior F did not occur in Solaris 10. When out-of-order data exists at the data sender, thus sending both data payload and SACKs, every other segment carries only bytes equal to SACK information appended (at most 36 bytes). This misbehavior halves the throughput for the time out-of-order data exists at the receive buffer, and is the typical web browsing scenario described above. We consider the misbehavior significant, and needs to be fixed.

Misbehavior G is observed on Solaris 10 and OpenSolaris. We ran the Test G multiple times with different time intervals X = {1, 5, 15} minutes. Even after 15 minutes, we frequently observed the reappearance of SACK blocks from a prior connection in later connections. The SACK based loss recovery algorithm does not work efficiently, when the TCP implementation has this misbehavior. For example, when two connections have overlapping sequence numbers, the latter connection sends a SACK for a data block that was never received. This will cause a decrease in throughput.

One time, we ran all the seven TBIT tests continuously on Solaris 10 and OpenSolaris machines, and noticed a scenario where a SACK block of the first connection in Test A appeared in the SYN-ACK segment of the *third* connection established in Test C. One time, all TBIT tests were executed and then repeated 45 minutes later. Even after 45 minutes, we observed an instance where the SACK block of Test E from the first set appeared in the SYN-ACK segment of Test E in the second set. We could not repeat this misbehavior with any regularity. Having a sender think data is acknowledged when in fact the data has not been received results in an inconsistent (i.e., unreliable) state. Fortunately, this misbehavior is corrected in Solaris 11.

5. RELATED WORK

Two methodologies are mainly used to infer the TCP behavior: passive and active measurements. In passive measurements, collected trace files are analyzed offline to infer a specific protocol behavior. In 1997 Paxson [14] presents *tcpanaly,* a tool which automatically analyses the correctness of TCP implementations by inspecting traces collected for bulk data transfers.

In 2001 Padhye et al. [13] describe the active measurement tool TBIT and its architecture. A number of TBIT tests are provided by authors including testing a remote web server's support for SACK, and testing if SACKs are correctly processed by web servers when retransmitting segments during SACK-based loss recovery. The authors reported that 41% of the web servers were SACK-enabled. Of the SACK-enabled web servers, 42% were tested to properly process SACKs.

In 2004 Ladha et al. [10] extended TBIT tests to measure the deployment of further TCP enhancements such as limited transmit, appropriate byte counting (ABC), early retransmit and SACK. The authors report that while 69% of tested web servers advertise being SACK-enabled, only 90 out of 344 (26%) actually process SACK information to properly perform sender side loss recovery.

In 2005 Medina et al. [12] follow up on [13] and investigate the correctness of TCP implementations. Active measurements using TBIT confirm that in 2004 that 68% of web servers tested were SACK-enabled up from the 41% reported in 2001 [13]. The authors found roughly 90% of the SACK-enabled web servers make use of information in SACKs that they receive, a significant increase from [10]. The authors also tested the generation of SACKs by web servers. Only one misbehavior is reported - where .5% of tests resulted in SACKs whose sequence numbers were shifted. The authors suggest plausible causes as buggy TCP implementations or middleboxes (NATs, fingerprint scrubbers). In our CAIDA traces thus far analyzed and our TBIT tests, we did not see any shifted sequence numbers. Note that our experimental design has no middleboxes.

Our research combines both methodologies. We use TBIT to create synthetic TCP traffic to verify the proper SACK generation of TCP stacks. In addition, we capture TCP segments using tcpdump or wireshark for offline SACK generation analysis.

6. CONCLUSION

In this research, we designed a methodology and verified conformant SACK generation on 29 TCP stacks for a wide range of OSes: FreeBSD, Linux, Mac OS X, OpenBSD, Solaris and Windows. We identified the characteristics of the seven misbehaviors, and designed seven new TBIT tests to uncover these misbehaviors.

For the first five misbehaviors which are observed in the CAIDA trace files, we found at least one misbehaving TCP stack. We report various versions of OpenBSD and Windows OS to have misbehaving SACK generation implementations. In general, the misbehaving SACK implementations can cause a less efficient SACK-based loss recovery which yields to decreased throughput and longer transfer times.

During the TBIT testing, we identified two additional misbehaviors (F and G). Misbehavior F decreases the throughput by sending less than expected data while using SACKs. Most Linux and OpenSolaris systems show this misbehavior. Misbehavior G is more serious and can cause a TCP connection to be inconsistent should the sequence number space of one connection overlap that of a prior connection. Solaris 10 and OpenSolaris systems misbehave in this manner.

We note that for all misbehaviors, because SACKs are advisory thus allowing a data receiver to renege on all SACKed out-of-order data, eventually the data sender-receiver will timeout, discard all SACK information, and return to a correct state. Thus the data flow remains reliable; only performance degradation may occur.

As stated in the Introduction, we discovered SACK misbehaviors during our investigation of data reneging [7]. In that investigation, we argue that SACKs should be "permanent" (not advisory) meaning a data receiver MUST NOT renege on out-of-order data. If SACKs were to become permanent, since misbehavior G can result in unreliable data transfer, it would have to be fixed. While we hope misbehaviors A-F will be fixed, even if left as is, they will only result in reduced performance, not unreliable protocol behavior.

While simple in concept, SACK handling is complex to implement.

7. ACKNOWLEDGMENTS

The authors thank Jonathan Leighton, Aasheesh Kolli and Ersin Ozkan for their valuable discussions and comments while developing this paper. The authors also sincerely thank the anonymous reviewers for their constructive feedback.

8. REFERENCES

[1] M. Allman, C. Hayes, H. Kruse, S. Ostermann, "TCP performance over satellite links", 5th International Conference on Telecommunications Systems, 3/97

[2] Apache HTTP Server, httpd.apache.org/

[3] E. Blanton, M. Allman, K. Fall, L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC3517, 4/03

[4] S. Bradner, "Key words for use in RFCs to indicate requirement levels", RFC2119, 3/9

[5] R. Bruyeron, B. Hemon, L. Zhang, "Experimentations with TCP selective acknowledgment", ACM Computer Communication Review, 28(2), 4/98, pp. 54-77

[6] CAIDA Internet Data – Passive Data Sources, www.caida.org/data/passive/

[7] N. Ekiz, "Transport layer reneging," PhD Dissertation, CISC Dept., Univ of Delaware (in progress)

[8] K. Fall, S. Floyd, "Simulation-based comparisons of Tahoe, Reno, and SACK TCP", ACM Computer Communication Review, 26(3), 6/96, pp. 5-21

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC2616, 6/99

[10] S. Ladha, P. D. Amer, A. J. Caro, J. R. Iyengar, "On the prevalence and evaluation of recent TCP enhancements", IEEE Globecom, 11/04

[11] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgment Options", RFC2018, 10/96

[12] A. Medina, M. Allman, S. Floyd, "Measuring the evolution of transport protocols in the Internet", ACM SIGCOMM Computer Communication Review, 4/05

[13] J. Padhye, S. Floyd, "On inferring TCP behavior", ACM SIGCOMM, 8/01, pp. 287-298

[14] V. Paxson, "Automated packet trace analysis of TCP implementations", ACM SIGCOMM, 9/97

[15] J. Postel, "Transmission Control Protocol", RFC793, 9/81

[16] R. Stewart, "Stream Control Transmission Protocol", RFC4960, 9/07

[17] TBIT tests and TBIT packet captures, pel.cis.udel.edu/tbit-tests/

[18] Tcpdump, www.tcpdump.org/1999

[19] The TCP Behavior Inference Tool, www.icir.org/tbit/

[20] VirtualBox, www.virtualbox.org/

[21] Wireshark, www.wireshark.org/