CS588 Research Project
School of Engineering and Applied Science
University of Virginia

Analysis of the Feasibility of Keystroke Timing Attacks Over SSH Connections

Submitted by:

Michael Augustus Hogye
Christopher Thaddeus Hughes
Joshua Michael Sarfaty
Joseph David Wolf

CS 588 – Cryptology
December 5, 2001

Table of Contents

1. Introduction....1
2. Related Work....2
  2.1 Relevant SSH Protocol Information....2
  2.2 Keystroke Timing Attacks are a Practical Threat....3
  2.3 Keystroke Timing Attacks are *not* a Practical Threat....5
  2.4 Relevance....6
  2.5 Solutions....7
  2.6 Problems in [SWT01]....7
  2.7 Measuring Keystroke Timing over a Network....8
    2.7.1 Methodology....8
    2.7.2 Data Analysis....9
  2.8 Packet Sniffing Techniques....10
    2.8.1 Methodology....10
    2.8.2 Data Analysis....11
3. Evaluation....12
4. Conclusion....12
5. References....14
Appendix A – `su` signatures....15
  SSH-2: littlenerd.dorm to blue.unix....15
  SSH-2: littlenerd.dorm to sgi-8....16
  SSH-2: blue.unix to littlenerd.dorm....17
Appendix B – keystroke-timing client & server code....18
  ktclient.cpp....18
  ktserver.cpp....21

1. Introduction

Secure Shell (SSH) is a protocol for secure remote access across an insecure network. Information transferred using SSH is encrypted using strong encryption algorithms, providing strong protection from eavesdroppers on the network. Recent studies, however, have shown potential weaknesses in SSH that allow eavesdroppers to discover transmitted passwords more easily. These weaknesses are exploited by measuring the time between password keystrokes sent by SSH and using that information to narrow in on what the password might be—a procedure known as a keystroke timing attack [SWT01]. If these weaknesses can be successfully exploited, the security of computers using SSH servers will be severely compromised. Song, Wagner, and Tian, in their paper "Timing Analysis of Keystrokes and Timing Attacks on SSH," have claimed that these weaknesses are real and threatening for SSH [SWT01]. SSH Communications Security, on the other hand, insists that exploiting these weaknesses is an impossibility in their article "Timing Analysis is Not a Real-Life Threat to SSH Secure Shell Users." This project seeks to determine who is correct.

2. Related Work

The following sections provide relevant background information on SSH and present cases for both sides of the SSH keystroke timing attack practicality debate.

2.1 Relevant SSH Protocol Information

Secure Shell (SSH) was invented by Tatu Ylönen in 1995. It existed in this form (SSH-1) until late 1997 / early 1998 when an Internet Draft was submitted for SSH-2 by the Internet Engineering Task Force (IETF) [OVER01]. The SSH-2 protocol is an entire rewrite of the original one, and it contains many differences. Some of these differences have a marked effect on the feasibility of keystroke timing attacks.

Both the SSH-1 and the SSH-2 protocols use public key technology to establish a session key. The session key is used to encrypt and decrypt packets using a symmetric cipher. The table below shows the format of both SSH-1 and SSH-2 packets in the correct order.

SSH-1				**SSH-2**		
Field	*Size (bytes)*	*Encrypted ?*		*Field*	*Size (bytes)*	*Encrypted ?*
Length of Data Payload (N)	4	No		Length of Packet (M)	4	Yes
Random Padding		Yes		Length of Random Padding (P)	1	Yes
Packet Type	1	Yes		Data Payload	M – P – 1	Yes
Data Payload	N	Yes		Random Padding	P	Yes
MAC	Mac length	No		MAC	Mac length	No

[YKM+01,OVER01]

The Random Padding fields serve to pad the encrypted information in the packets to either the block length of the symmetric cipher or 8 bytes, whichever is greater.

The main difference that affects keystroke timing attacks is that in SSH-1, the Length of Data Payload field is not encrypted, whereas in SSH-2 it is. Another difference that allows SSH-2 to be more secure in general is the stronger data integrity checks it uses, where SSH-1 relies on a cyclic redundancy check (CRC) [YL01]. A feature in both versions of SSH that allows keystroke timing attacks to be plausible is the use of immediate mode. Immediate mode sends each keystroke to the server immediately in its own packet. This allows the interactive user experience, and the ability to edit text on-screen with the use of backspace, for example.

2.2 Keystroke Timing Attacks are a Practical Threat

In "Timing Analysis of Keystrokes and Timing Attacks on SSH," the authors claim that SSH leaks password information in two ways. First, SSH transmits password packets that are padded to an eight-byte boundary during the login. From this, an eavesdropper can learn the approximate size of the password by looking at the number of packets that are sent. If only one packet is sent, then the eavesdropper knows the password is at most 7 characters long.

Secondly, when SSH is in interactive mode, it is useful for eavesdroppers to monitor the time intervals between sent packets [ZP00]. SSH automatically goes into interactive mode after the initial login. During this mode, eavesdroppers can determine which packets contain password characters by carefully monitoring traffic between the client and the server. Passwords are sent during interactive mode on two occasions: when a switch user command is executed and when the client starts a nested SSH session. The client sends the server an `su` command (`su` stands for "switch user"), followed by a return, prior to the transmission of a password [SWT01]. After the `su` command is acknowledged, the next packets sent contain the password characters. These packets are

not echoed by the server, thus indicating to the eavesdropper that these packets contain the characters of the password. Figure 1 shows an SSH packet transfer between the client and server for an `su` command execution. These transfers form a "signature" for recognizing the `su` command. The numbers indicate the packet size in bytes. Notice how the characters of the password "Julia" are not echoed by the server, indicating that those packets contain password characters.

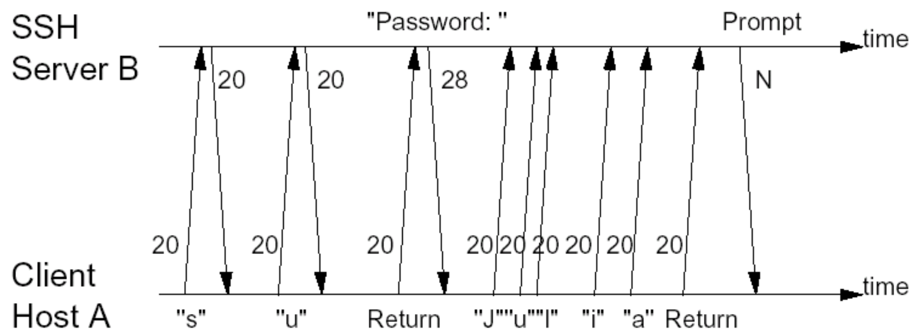


Figure 1 – identifying password character packets sent after an `su` command in SSH-1 [SWT01].

Additionally, [SWT01] argued that password character packets can be identified if a nested SSH session is started within a currently open session. Figure 2 shows such a nested SSH session. Although the second SSH session between B and C sends the password in one large chunk, the characters of the password are sent in separate packets from A to B. This allows eavesdroppers to sniff these packets and obtain information about the password used in the second session.

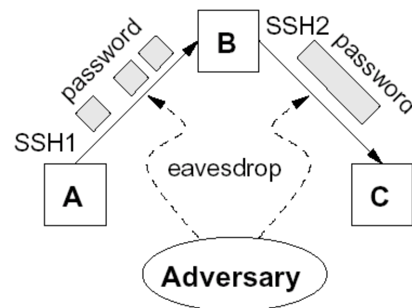


Figure 2 – nested SSH attack [SWT01].

Once the password character packets have been identified, keystroke timing analysis can be performed on the time intervals to aid in the cracking of the passwords. These time intervals can reveal information about the password typed. Using a statistically complex procedure involving Hidden Markov Models and *n*-Viterbi

algorithms (the details of which are beyond the scope of this paper), [SWT01] estimated that an attacker "could potentially extract 1.2 bits per character pair" when keystroke latency information is known for all possible character pairs present in the password. They also argued that since the entropy of English is very low, the "1.2-bit information gain per character pair leaked through the latency information seems to be significant." To evaluate these arguments, the team implemented a program called Herbivore that attempted keystroke timing attacks on passwords. Herbivore was able, in their experiments, to reduce the search space for randomly chosen 8-character passwords by a factor of 50, thus making the password more vulnerable to brute force attacks.

The authors concluded that the two weaknesses "reveal a surprising amount of information on passwords and other text typed over SSH sessions" and as a consequence, "SSH is not as secure as commonly believed." Others, particularly Yin Zhang and Vern Paxson in "Detecting Backdoors," have presented support for using timing analysis to gain information on passwords [ZP00]. Solar Designer and Dug Song have also stated that SSH is vulnerable to keystroke timing attacks in a posted message on Bugtraq in March 2001 [SD01].

2.3 Keystroke Timing Attacks are *not* a Practical Threat

The loudest opposition to the previous findings come from SSH Communications Security, the makers of SSH Secure Shell software . The following is the company's statement on keystroke timing attack weaknesses in SSH [SSH01a]:

> The attack presented by [Song, Wagner, and Tian] does not pose a practical threat to SSH Secure Shell users because:
>
> - SSH Secure Shell transmits the user's normal login password in one encrypted packet. Thus, timing the individual characters by monitoring the encrypted traffic is not possible.

- Determining the length of the packet as described in the article is not possible, because SSH Secure Shell 3.0.0 and later pad the password packet so that its length cannot be determined.
- If a password to another application or server is typed over an established SSH Secure Shell connection, performing timing analysis on this password is theoretically possible. However, it is not practical in reality, because:
    - The attack requires reliable per-user reference data on keystroke timings, which requires co-operation from the user.
    - Determining where a password starts in an encrypted connection is probabilistic, and the analysis is confused by falsely guessed password locations in the session.
    - According to our analysis, performing the attack on a realistic 8-character password with unrestricted character set would require approximately 120 terabytes (120 000 gigabytes) of memory, which is not feasible with the technology available in the next several years.
- Even if someone was able to successfully perform the attack, it would only reduce the work factor for trying all possible passwords by a factor of 50, which corresponds to shortening the password by approximately one character (i.e., an 8-character password would become effectively a 7-character password which would still have to be guessed correctly).

Although Solar Designer and Dug Song have sided with those who believe in the practicality of keystroke timing attacks, they do agree with SSH Secure Communications in some respects. Most notably, they admit that Version 2 of the SSH protocol eliminated the weakness of revealing approximate password length during the login phase [SD01]. They also mentioned that several SSH vendors have implemented fixes in their programs that strengthen the alleged SSH weaknesses.

2.4 Relevance

The information presented by both sides of the debate is relevant to the conclusion this paper makes on the practicality of keystroke timing attacks. Any significant points, data, and assumptions made by either side must be scrutinized carefully in order to develop reliable conclusions. For example, the fact that SSH Secure Communications has a vested interest in maintaining the integrity of their software is taken into consideration, as well as [SWT01]'s usage of eight-year-old internet latency

statistics and the large amount of password data they gathered prior to attempting keystroke attacks.

2.5 Solutions

In order to determine whether keystroke timing attacks are a legitimate threat to SSH connections in the real world, we decided to investigate some of the ways an attacker might try to determine information about an SSH session. To do this, we examined the feasibility of determining keystroke timing information over a network by writing simple client and server programs to measure keystroke and packet timing intervals. We also used packet sniffing techniques to examine an SSH session in order to find an `su` command and determine which packets contained the password. First, however, we examined [SWT01]'s claims more carefully.

2.6 Problems in [SWT01]

The research done in [SWT01] had the following problems that suggested that keystroke timing attacks are not completely practical to execute.

1. A random password was typed repeatedly so Herbivore could work with reliable keystroke timing information. In reality, SSH attackers would not be provided with this large amount of information.
2. Herbivore only attempted to crack random passwords chosen from a finite character space that consisted of combinations of only 10 letters and 5 numbers—142 character pairs total. This comprises only 3% of the total 4,278 possible pairs of letters, numbers, and special characters than may be present in a realistic password.
3. The network delay and Internet lag statistics used in the calculations were taken from a 1993 report. As a result, delay was ignored in their experiments, despite the fact

that current Internet latencies, which can reach over 170ms, are significant.

4. [SWT01] used SSH-1 in their experiments. SSH-1 has the notorious weakness of not encrypting the payload length field in the header. This field only records length of the actual payload data, not the padding that is added to the packets. With this information, it is easier to recognize `su` signatures (see Figure 1). Using SSH-2, however, obtaining the length of the actual data is much more difficult, if not impossible, because SSH-2 encrypts the payload length.

2.7 Measuring Keystroke Timing over a Network

2.7.1 Methodology

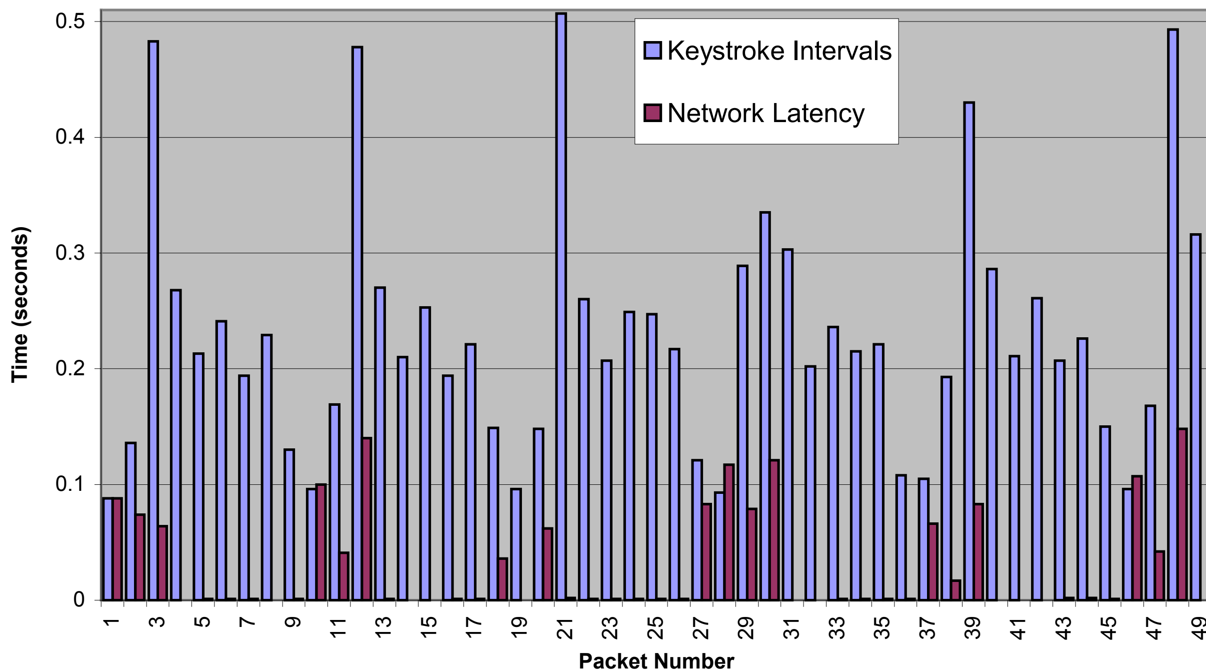
In order to determine precise keystroke timing information over a real network, we first developed simple client and server programs. When the server is run on a machine, it waits for connections from the client. When the client is run from another machine, it establishes a connection with the server socket. (By specifying a preprocessor directive in the code, we can set the programs to use SSL or simple, unencrypted sockets.) Then, whenever a key is pressed on the client, that key is sent to the server, along with the time difference since the last key was pressed. When the server receives one of these packets, it prints the key, the received time difference, and the time difference since the last packet it received from the client. This allows us to observe the timing difference between adjacent keystrokes, both on the client side, where the timing can be measured without network delay, and on the server side, where the time difference is a function of both the actual keystroke timing and the network delay.

In order to see how network traffic influenced our ability to observe keystroke timing parameters, we performed this experiment several times, using different machines

and at different times of the day.

2.7.2 Data Analysis

After examining the time delays introduced by the network, it was apparent that these delays would make SSH keystroke timing attacks very difficult. As Figure 3 demonstrates, the network latency can be relatively large in comparison to the inter-keystroke timing. However, having a large latency will not thwart the keystroke timing attack. If the latency is constant, it will merely add a constant offset to the packet times,



leaving the times between packets the same as the times between keystrokes.

The average interval between keystrokes when typing a password is around 208 ms. The average network latency for these packets is 1.8 ms. However, the latency varies widely, having a standard deviation of approximately 48.1ms. By introducing a somewhat random delay, the latency will change the packet arrival times and thus alter the perceived inter-keystroke timing. Because keystroke timing attacks rely on precise

information about the inter-keystroke timing intervals, this variance makes performing statistical analysis exceedingly difficult. This point is moot for a sniffer running on the same network node as the client, but that situation is much less useful to an attacker and a keystroke timing attack would likely yield no useful information.

2.8 Packet Sniffing Techniques

2.8.1 Methodology

In order to isolate the user's password in a TCP stream, one can attempt to detect the SSH signature of the UNIX `su` command and the following password. The `su` command signature is presented in Figure 2. In order to test the feasibility of this attack, we attempted to implement the scheme presented in Song using the packet sniffer Ethereal. Ethereal is a generic packet sniffer which sets the Ethernet card to promiscuous mode, allowing it to report all network traffic on the local subnet.

We began an SSH session between our local machine and one of the nodes in UVA's 'blue.unix' cluster. We then started Ethereal on the local machine, allowing us to examine all SSH traffic in its encrypted format between the two machines.

Once we had the ability to observe the encrypted packets between our local machine and the 'blue.unix' node, we then performed several `su` commands, observing the traffic generated by each keystroke. Because the `su` command is executed in SSH's interactive mode, each key-press generates a packet to be sent to the 'blue.unix' host node. Then the host acknowledges and echoes each keystroke back to the client, and finally, the client acknowledges the character it received, printing the character on the screen. However, passwords are never printed on the screen, so the packets containing

the characters of the password are sent to host, but are not echoed back to the client. The packets which are not echoed correspond to the characters of the password.

We performed this experiment between several different client / server pairs, using both the SSH-1 and SSH-2 protocols.

2.8.2 Data Analysis

By examining the SSH session packet logs for each scenario, we found that it is often very difficult to determine when the `su` command and the following password occur in the packet stream. In particular, the SSH session packet stream depends largely on the implementation of the SSH server used in the connection. The `su` signatures from various client / server pairs can be seen in Appendix A.

When connecting to 'blue.unix,' which uses the SSH Secure Shell v2.4.0 server, there was a clear pattern to indicate the presence of an `su` command in the packet stream. The 's,' 'u,' and return characters are echoed by the server, and then the password prompt is sent. Then the client sends an un-echoed stream of characters representing the password. Someone searching the session for this pattern could very easily identify the `su` command.

However, the machine 'sgi-8,' which uses the newer SSH Secure Shell v3.0.1, has no clear pattern to indicate the presence of an `su` command. Each character sent in immediate mode generates four packets. First the character is sent from client to server. Then it is acknowledged by the server. Then it is echoed by the server. Finally, the client acknowledges the echoed character. The problem in detecting as signature arises because this sequence is the same whether the packet is echoed or not. When the actual password characters are being typed, the server still sends dummy packets to take the place of an echoed character. Furthermore, every packet containing data has the same

packet length. This prevents the identification of the `su` command signature. Likewise, the data in Appendix A for 'littlenerd.dorm,' which uses OpenSSH v2.9, follows a similar pattern.

We have determined that the implementation of the SSH server plays a very large role in determining whether the `su` command has a detectable signature. The SSH protocol allows the server to easily send dummy packets which obfuscate the pattern. Whether or not the SSH server does this plays a large role in its security.

3. Evaluation

Our goal was to determine the efficacy of keystroke timing attacks on encrypted SSH sessions. To this end, we performed several experiments to estimate the vulnerability of SSH in the real world. The experiments we performed are very representative of the methodology an attack would use. In addition, our experiments were performed using modern, real-life equipment as opposed to laboratory conditions. Our data takes into account the parameters of these real-world scenarios. For example, Song's research assumes an average Internet latency of 10 ms [SWT01]. We found that the latency could likely be much larger, and that latency variance is more important than average latency when performing keystroke timing attacks.

Even though we used real-world networks and equipment, the scenarios examined in our experiments are a very small subset of all the scenarios in which SSH may be used. For this reason, it would be helpful to perform our SSH keystroke analysis experiments on a wider variety of networks, with many different clients and servers.

4. Conclusion

Our findings indicate that network latency on a network with any significant

traffic is enough to obscure any significant keystroke timing information. Furthermore, the variation in SSH protocol implementations make it difficult to find the signatures of specific commands typed in an interactive SSH session. It is particularly hard to do this for some SSH servers, because these servers send dummy packets to fake an echo for every character.

We are not claiming that SSH is a totally secure means for communication – it is impossible to determine for certain that a protocol is entirely safe. In contrast, a protocol can be demonstrated insecure by finding only one vulnerability. We are merely claiming that the keystroke timing vulnerabilities found so far in the SSH protocol are mostly of academic interest, and difficult to apply in the real world.

5. References

[FAQ01] SSH FAQ – Section 1. http://www.inf.bme.hu/~mulzs/sshfaq/ssh-faq-1.html#ss1.8, 2001.

[OVER01] Van Dyke Software Homepage. http://www.vandyke.com/solutions/ssh_overview/ssh_overview_history.html, 2001.

[SD01] Solar Designer and Dug Song. Passive Analysis of SSH (Secure Shell) Traffic. Openwall advisory, OW-003. `http://security-archive.merton.ox.ac.uk/bugtraq-200103/0258.html`, March 2001.

[SSH01a] Timing Analysis is Not a Real-Life Threat to SSH Secure Shell Users. `http://www.ssh.com/products/ssh/timing_analysis.cfm`, 2001.

[SWT01] Dawn Song, David Wagner, Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *Proc. Of 10th USENIX Security Symposium*, August 2001.

[YKM+01] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen; "SSH Transport Layer Protocol", http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-11.txt, 2001.

[YL01] T. Ylonen, S. Lehtinen; "SSH File Transfer Protocol", http://search.ietf.org/internet-drafts/draft-ietf-secsh-filexfer-02.txt, 2001.

[ZP00] Yin Zhang and Vern Paxson. Detecting Backdoors. In *Proc. Of 9th USENIX Security Symposium*, August 2000.

Appendix A – `su` signatures

SSH-2: littlenerd.dorm to blue.unix

Time	Source	Destination	Flags	Length
3.098878	littlenerd.dorm	blue.unix	[PSH,ACK]	40
3.101986	blue.unix	littlenerd.dorm	[PSH,ACK]	40
3.102006	littlenerd.dorm	blue.unix	[ACK]	0
5.745584	littlenerd.dorm	blue.unix	[PSH,ACK]	40
5.748593	blue.unix	littlenerd.dorm	[PSH,ACK]	40
5.748619	littlenerd.dorm	blue.unix	[ACK]	0
7.311973	littlenerd.dorm	blue.unix	[PSH,ACK]	40
7.315193	blue.unix	littlenerd.dorm	[PSH,ACK]	40
7.315209	littlenerd.dorm	blue.unix	[ACK]	0
7.400233	blue.unix	littlenerd.dorm	[PSH,ACK]	48
7.400246	littlenerd.dorm	blue.unix	[ACK]	0
7.401544	blue.unix	littlenerd.dorm	[PSH,ACK]	48
7.401556	littlenerd.dorm	blue.unix	[ACK]	0
11.938379	littlenerd.dorm	blue.unix	[PSH,ACK]	40
12.111122	blue.unix	littlenerd.dorm	[ACK]	0
12.303909	littlenerd.dorm	blue.unix	[PSH,ACK]	40
12.321037	blue.unix	littlenerd.dorm	[ACK]	0
12.77418	littlenerd.dorm	blue.unix	[PSH,ACK]	40
12.950947	blue.unix	littlenerd.dorm	[ACK]	0
13.234006	littlenerd.dorm	blue.unix	[PSH,ACK]	40
13.371064	blue.unix	littlenerd.dorm	[ACK]	0
13.628537	littlenerd.dorm	blue.unix	[PSH,ACK]	40
13.780939	blue.unix	littlenerd.dorm	[ACK]	0
16.450322	littlenerd.dorm	blue.unix	[PSH,ACK]	40
16.460849	blue.unix	littlenerd.dorm	[PSH,ACK]	40
16.460864	littlenerd.dorm	blue.unix	[ACK]	0
16.495927	blue.unix	littlenerd.dorm	[PSH,ACK]	56
16.495938	littlenerd.dorm	blue.unix	[ACK]	0
16.498394	blue.unix	littlenerd.dorm	[PSH,ACK]	64
16.498405	littlenerd.dorm	blue.unix	[ACK]	0
16.499729	blue.unix	littlenerd.dorm	[PSH,ACK]	88
16.499741	littlenerd.dorm	blue.unix	[ACK]	0

SSH-2: littlenerd.dorm to sgi-8

Time	Source	Destination	Flags	Length
2.3337	littlenerd.dorm	sgi-8	[PSH,ACK]	48
2.335359	sgi-8	littlenerd.dorm	[ACK]	0
2.33744	sgi-8	littlenerd.dorm	[PSH,ACK]	48
2.337454	littlenerd.dorm	sgi-8	[ACK]	0
2.874851	littlenerd.dorm	sgi-8	[PSH,ACK]	48
2.876256	sgi-8	littlenerd.dorm	[ACK]	0
2.877779	sgi-8	littlenerd.dorm	[PSH,ACK]	48
2.877793	littlenerd.dorm	sgi-8	[ACK]	0
12.208276	littlenerd.dorm	sgi-8	[PSH,ACK]	48
12.209923	sgi-8	littlenerd.dorm	[ACK]	0
12.211717	sgi-8	littlenerd.dorm	[PSH,ACK]	48
12.21173	littlenerd.dorm	sgi-8	[ACK]	0
12.278076	sgi-8	littlenerd.dorm	[PSH,ACK]	48
12.27809	littlenerd.dorm	sgi-8	[ACK]	0
13.87318	littlenerd.dorm	sgi-8	[PSH,ACK]	48
13.87456	sgi-8	littlenerd.dorm	[ACK]	0
14.145405	littlenerd.dorm	sgi-8	[PSH,ACK]	48
14.147019	sgi-8	littlenerd.dorm	[ACK]	0
14.396384	littlenerd.dorm	sgi-8	[PSH,ACK]	48
14.398676	sgi-8	littlenerd.dorm	[ACK]	0
14.650641	littlenerd.dorm	sgi-8	[PSH,ACK]	48
14.651963	sgi-8	littlenerd.dorm	[ACK]	0
14.883592	littlenerd.dorm	sgi-8	[PSH,ACK]	48
14.885102	sgi-8	littlenerd.dorm	[ACK]	0
15.124763	littlenerd.dorm	sgi-8	[PSH,ACK]	48
15.126104	sgi-8	littlenerd.dorm	[ACK]	0
15.371296	littlenerd.dorm	sgi-8	[PSH,ACK]	48
15.372559	sgi-8	littlenerd.dorm	[ACK]	0
15.628701	littlenerd.dorm	sgi-8	[PSH,ACK]	48
15.630228	sgi-8	littlenerd.dorm	[ACK]	0

SSH-2: blue.unix to littlenerd.dorm

Time	Source	Destination	Flags	Length
1.752113	blue.unix	littlenerd.dorm	[PSH,ACK]	52
1.752405	littlenerd.dorm	blue.unix	[PSH,ACK]	52
1.773471	blue.unix	littlenerd.dorm	[ACK]	0
2.05564	blue.unix	littlenerd.dorm	[PSH,ACK]	52
2.055825	littlenerd.dorm	blue.unix	[PSH,ACK]	52
2.193668	blue.unix	littlenerd.dorm	[ACK]	0
2.279993	blue.unix	littlenerd.dorm	[PSH,ACK]	52
2.280624	littlenerd.dorm	blue.unix	[PSH,ACK]	52
2.286287	littlenerd.dorm	blue.unix	[PSH,ACK]	52
2.401127	blue.unix	littlenerd.dorm	[ACK]	0
4.090947	blue.unix	littlenerd.dorm	[PSH,ACK]	52
4.091171	littlenerd.dorm	blue.unix	[PSH,ACK]	52
4.272255	blue.unix	littlenerd.dorm	[ACK]	0
4.373262	blue.unix	littlenerd.dorm	[PSH,ACK]	52
4.373389	littlenerd.dorm	blue.unix	[PSH,ACK]	52
4.473753	blue.unix	littlenerd.dorm	[ACK]	0
4.923786	blue.unix	littlenerd.dorm	[PSH,ACK]	52
4.924057	littlenerd.dorm	blue.unix	[PSH,ACK]	52
5.103766	blue.unix	littlenerd.dorm	[ACK]	0
5.265581	blue.unix	littlenerd.dorm	[PSH,ACK]	52
5.265704	littlenerd.dorm	blue.unix	[PSH,ACK]	52
5.313938	blue.unix	littlenerd.dorm	[ACK]	0
5.49171	blue.unix	littlenerd.dorm	[PSH,ACK]	52
5.491922	littlenerd.dorm	blue.unix	[PSH,ACK]	52
5.52352	blue.unix	littlenerd.dorm	[ACK]	0
5.602852	blue.unix	littlenerd.dorm	[PSH,ACK]	52
5.602933	littlenerd.dorm	blue.unix	[PSH,ACK]	52
5.733511	blue.unix	littlenerd.dorm	[ACK]	0
5.826174	blue.unix	littlenerd.dorm	[PSH,ACK]	52
5.826256	littlenerd.dorm	blue.unix	[PSH,ACK]	52
5.943385	blue.unix	littlenerd.dorm	[ACK]	0
6.049279	blue.unix	littlenerd.dorm	[PSH,ACK]	52
6.049366	littlenerd.dorm	blue.unix	[PSH,ACK]	52
6.14648	blue.unix	littlenerd.dorm	[ACK]	0
6.222497	blue.unix	littlenerd.dorm	[PSH,ACK]	52
6.222585	littlenerd.dorm	blue.unix	[PSH,ACK]	52
6.353631	blue.unix	littlenerd.dorm	[ACK]	0
6.387635	blue.unix	littlenerd.dorm	[PSH,ACK]	52
6.387844	littlenerd.dorm	blue.unix	[PSH,ACK]	52
6.387932	littlenerd.dorm	blue.unix	[PSH,ACK]	52
6.563442	blue.unix	littlenerd.dorm	[ACK]	0
6.563494	littlenerd.dorm	blue.unix	[PSH,ACK]	152
6.773544	blue.unix	littlenerd.dorm	[ACK]	0

Appendix B – keystroke-timing client & server code

ktclient.cpp

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <sys/timeb.h>

#define USE_SSL

#ifdef USE_SSL

#include <openssl/ssl.h>
#include <openssl/err.h>

#endif



int main(int argc, char **argv)
{
  int port;
  int result = 0;
  struct hostent *server_name = 0;

  if ((argc != 3) || ((server_name = gethostbyname(argv[1])) == 0)
      || (sscanf(argv[2], "%d", &port) != 1))
  {
    printf("Usage: %s <ip addr.> <port no.>\n", argv[0]);
    return 1;
  }

  int sd, accepted_sd;
  socklen_t client_size;
  struct sockaddr_in client_addr, server_addr;

#ifdef USE_SSL
  OpenSSL_add_all_algorithms();
  SSL_load_error_strings();
  SSL_library_init();

  fprintf(stderr, "Getting SSL context ... ");
  SSL_CTX *ssl_context = SSL_CTX_new(SSLv23_client_method());
  if (!ssl_context)
  {
    fprintf(stderr, "failed:\n\t%s.\n", ERR_error_string(ERR_get_error(), 0));
    return 1;
  }
  fprintf(stderr, "success.\n");

  fprintf(stderr, "Getting SSL ... ");
  SSL *ssl = SSL_new(ssl_context);
  if (!ssl)
  {
    fprintf(stderr, "failed:\n\t%s.\n", ERR_error_string(ERR_get_error(), 0));
    return 1;
```

```
  }
  fprintf(stderr, "success.\n");
#endif


  // create socket
  fprintf(stderr, "Creating socket ... ");
  sd = socket(AF_INET, SOCK_STREAM, 0);
  if (sd < 0)
  {
    fprintf(stderr, "failed.\n");
    return 1;
  }
  fprintf(stderr, "success.\n");

  // bind port
  fprintf(stderr, "Binding port %d ... ", port);

  server_addr.sin_family = server_name->h_addrtype;
  memcpy((char *) &server_addr.sin_addr.s_addr,
         server_name->h_addr_list[0], server_name->h_length);
  server_addr.sin_port = htons(port);

  client_addr.sin_family = AF_INET;
  client_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  client_addr.sin_port = htons(0);
  if (bind(sd, (struct sockaddr *) &client_addr,
      sizeof(client_addr)) < 0)
  {
    fprintf(stderr, "failed.\n");
    return 1;
  }
  fprintf(stderr, "success.\n");

  // connect
  fprintf(stderr, "Connecting ... ");
  result = connect(sd, (struct sockaddr *) &server_addr, sizeof(server_addr));
  if(result < 0)
  {
    fprintf(stderr, "failed.\n");
    return 1;
  }
  fprintf(stderr, "success.\n");

#ifdef USE_SSL
  fprintf(stderr, "Initializing SSL ... ");
  result = SSL_set_fd(ssl, sd);
  if (result != 1)
    fprintf(stderr, "failed:\n\t%s.\n", ERR_error_string(ERR_get_error(), 0));
  else
    fprintf(stderr, "success.\n");

  fprintf(stderr, "Awaiting secure connection ... ");
  result = SSL_connect(ssl);
  if (result != 1)
    fprintf(stderr, "failed:\n\t%s.\n", ERR_error_string(ERR_get_error(), 0));
  else
    fprintf(stderr, "success.\n");
#endif



  // enter raw mode so we can get each keystroke
  struct termios term_mode;
  tcgetattr(0, &term_mode);
  fprintf(stderr, "Entering raw mode ... ");
  term_mode.c_lflag &= ~ICANON;
```

```
  tcsetattr(0, 0, &term_mode);
  fprintf(stderr, "all set.\n");


  char ch;
  char buf[4097];
  int bytes_sent = 0;
  timeb t;
  double current_time, previous_time;

  ftime(&t);
  current_time = t.time + 0.001*t.millitm;

  do
  {
    previous_time = current_time;

    ch = getc(stdin);

    ftime(&t);
    current_time = t.time + 0.001*t.millitm;

    fprintf(stderr, " : %.3f\n", current_time - previous_time);
    sprintf(buf, "%c %.3f", ch, current_time - previous_time);

#ifdef USE_SSL
    bytes_sent = SSL_write(ssl, buf, strlen(buf));
#else
    bytes_sent = send(sd, (void *)(buf), strlen(buf), 0);
#endif
    if (bytes_sent < 0)
      fprintf(stderr, "Error.\n");

  } while (1);

#ifdef USE_SSL
  SSL_free(ssl);
#endif


  return 0;
}
```

ktserver.cpp

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/timeb.h>

#define USE_SSL

#ifdef USE_SSL

#include <openssl/ssl.h>
#include <openssl/err.h>

#endif


#define STDOUT stdout
#define ERROUT stderr

int main(int argc, char **argv)
{
  int port;
  int result = 0;

  if ((argc != 2) || (sscanf(argv[1], "%d", &port) != 1))
  {
    fprintf(ERROUT, "Usage: %s <port no.>\n", argv[0]);
    return 1;
  }

  int sd, accepted_sd;
  socklen_t client_size;
  struct sockaddr_in client_addr, server_addr;


  setbuf(STDOUT, 0);
  setbuf(ERROUT, 0);

#ifdef USE_SSL
  OpenSSL_add_all_algorithms();
  SSL_load_error_strings();
  SSL_library_init();

  SSL_CTX *ssl_context = SSL_CTX_new(SSLv23_server_method());
  if (!ssl_context)
  {
    fprintf(ERROUT, "Getting SSL context failed:\n\t%s.\n",
            ERR_error_string(ERR_get_error(), 0));
    return 1;
  }

  result = SSL_CTX_use_certificate_file(ssl_context, "server-req.pem",
                                        SSL_FILETYPE_PEM);
  if (result != 1)
  {
    fprintf(ERROUT, "Initializing certificate failed:\n\t%s.\n",
            ERR_error_string(ERR_get_error(), 0));
    return 1;
  }

  result = SSL_CTX_use_PrivateKey_file(ssl_context, "server-key.pem",
                                       SSL_FILETYPE_PEM);
```

```
  if (result != 1)
  {
    fprintf(ERROUT, "Initializing key failed:\n\t%s.\n",
            ERR_error_string(ERR_get_error(), 0));
    return 1;
  }

  SSL *ssl = 0;
#endif


  // create socket
  sd = socket(AF_INET, SOCK_STREAM, 0);
  if (sd < 0)
  {
    fprintf(ERROUT, "Creating socket failed.\n");
    return 1;
  }

  // bind port
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(port);
  if (bind(sd, (struct sockaddr *) &server_addr,
      sizeof(server_addr)) < 0)
  {
    fprintf(ERROUT, "Binding port failed.\n");
    return 1;
  }

  // listen
  result = listen(sd, 5);
  if (result != 0)
  {
    fprintf(ERROUT, "Starting to listen failed: %d.\n", result);
    return 1;
  }



  do
  {

    fprintf(STDOUT, "---------------------------------------\n");
    client_size = sizeof(client_addr);
    accepted_sd = accept(sd, (struct sockaddr *) &client_addr,
                         &client_size);
    if (accepted_sd < 0)
    {
      fprintf(ERROUT, "Waiting for connection failed.\n");
      return 1;
    }

#ifdef USE_SSL
    ssl = SSL_new(ssl_context);
    if (!ssl)
    {
      fprintf(ERROUT, "Initializing SSL failed:\n\t%s.\n",
ERR_error_string(ERR_get_error(), 0));
      return 1;
    }

    result = SSL_set_fd(ssl, accepted_sd);
    if (result != 1)
      fprintf(ERROUT, "Securing connection failed:\n\t%s.\n",
ERR_error_string(ERR_get_error(), 0));
```

```
    result = SSL_accept(ssl);
    if (result != 1)
      fprintf(ERROUT, "Accepting secure connection failed:\n\t%s.\n",
ERR_error_string(ERR_get_error(), 0));
#endif

    unsigned char *client_ip = (unsigned char *)&client_addr.sin_addr.s_addr;
    fprintf(STDOUT, "Connected to %d.%d.%d.%d.\n", client_ip[0], client_ip[1],
client_ip[2], client_ip[3]);

    char buf[4097];
    int bytes_read = 0;
    timeb t;
    double current_time, previous_time;

    ftime(&t);
    current_time = t.time + 0.001*t.millitm;

    do
    {
      previous_time = current_time;

#ifdef USE_SSL
      bytes_read = SSL_read(ssl, buf, sizeof(buf) - 1);
#else
      bytes_read = recv(accepted_sd, (void *)buf, sizeof(buf) - 1, 0);
#endif
      ftime(&t);
      current_time = t.time + 0.001*t.millitm;

      if (bytes_read > 0)
      {
       buf[bytes_read] = 0;

       if (buf[0] == '\n')
         buf[0] = ' ';

       fprintf(STDOUT, "%s : %.3f\n", buf,
              current_time - previous_time);
      }

    } while (bytes_read > 0);

    if (bytes_read != 0)
      fprintf(ERROUT, "Communication error.\n");

#ifdef USE_SSL
    SSL_free(ssl);
#endif

  } while (1);



  return 1;
}
```