USENIX Association

Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

© 2001 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Timing Analysis of Keystrokes and Timing Attacks on SSH*

Dawn Xiaodong Song David Wagner Xuqing Tian
University of California, Berkeley

Abstract

`SSH` is designed to provide a secure channel between two hosts. Despite the encryption and authentication mechanisms it uses, `SSH` has two weakness: First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which reveals the approximate size of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed, which leaks the inter-keystroke timing information of users' typing. In this paper, we show how these seemingly minor weaknesses result in serious security risks.

First we show that even very simple statistical techniques suffice to reveal sensitive information such as the length of users' passwords or even root passwords. More importantly, we further show that by using more advanced statistical techniques on timing information collected from the network, the eavesdropper can learn significant information about what users type in `SSH` sessions. In particular, we perform a statistical study of users' typing patterns and show that these patterns reveal information about the keys typed. By developing a Hidden Markov Model and our key sequence prediction algorithm, we can predict key sequences from the inter-keystroke timings. We further develop an attacker system, *Herbivore* , which tries to learn users' passwords by monitoring `SSH` sessions. By collecting timing information on the network, Herbivore can speed up exhaustive search for passwords by a factor of 50. We also propose some countermeasures.

In general our results apply not only to `SSH`, but also to a general class of protocols for encrypting interactive traffic. We show that timing leaks open a new set of security risks, and hence caution must be taken when designing this type of protocol.

*This research was supported in part by the Defense Advanced Research Projects Agency under DARPA contract N6601-99-28913 (under supervision of the Space and Naval Warfare Systems Center San Diego) and by the National Science foundation under grants FD99-79852 and CCR-0093337.

1 Introduction

Just a few years ago, people commonly used astonishingly insecure networking applications such as `telnet`, `rlogin`, or `ftp`, which simply pass all confidential information, including users' passwords, in the clear over the network. This situation was aggravated through broadcast-based networks that were commonly used (e.g., Ethernet) which allowed a malicious user to eavesdrop on the network and to collect all communicated information [CB94, GS96].

Fortunately, many users and system administrators have become aware of this issue and have taken countermeasures. To curb eavesdroppers, security researchers designed the Secure Shell (`SSH`), which offers an encrypted channel between the two hosts and strong authentication of both the remote host and the user [Ylö96, SSL01, YKS$^+$00b]. Today, `SSH` is quite popular, and it has largely replaced `telnet` and `rlogin`.

Many users believe that they are secure against eavesdroppers if they use `SSH`. Unfortunately, in this paper we show that despite state-of-the-art encryption techniques and advanced password authentication protocols [YKS$^+$00a], `SSH` connections can still leak significant information about sensitive data such as users' passwords. This problem is particularly serious because it means users may have a false confidence of security when they use `SSH`.

In particular we identify that two seemingly minor weaknesses of `SSH` lead to serious security risks. First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use). Therefore an eavesdropper can easily learn the approximate length of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`). We show in the paper that this property can enable the eavesdropper to learn the exact length of users' passwords. More importantly, as we have verified, the time it takes the operating system to send out the packet after the key press is in general negligible comparing to the inter-keystroke timing. Hence an eavesdropper can learn the precise inter-keystroke timings of users' typing from the arrival times of packets.

Experience shows that users' typing follows stable patterns[1]. Many researchers have proposed to use the duration of key strokes and latencies between key strokes as a biometric for user authentication [GLPS80, UW85, LW88, LWU89, JG90, BSH90, MR97, RLCM98, MRW99]. A more challenging question which has not yet been addressed in the literature is whether we can use timing information about key strokes to infer the key sequences being typed. If we can, can we estimate quantitatively how many bits of information are revealed by the timing information? Experience seems to indicate that the timing information of keystrokes reveals some information about the key sequences being typed. For example, we might have all experienced that the elapsed time between typing the two letters "er" can be much smaller than between typing "qz". This observation is particularly relevant to security. Since as we show the attacker can get precise inter-keystroke timings of users' typing in a SSH session by recording the packet arrival times, if the attacker can infer what users type from the inter-keystroke timings, then he could learn what users type in a SSH session from the packet arrival times.

In this paper we study users' keyboard dynamics and show that the timing information of keystrokes does leak information about the key sequences typed. Through more detailed analysis we show that the timing information leaks about 1 bit of information about the content per keystroke pair. Because the entropy of passwords is only 4–8 bits per character, this 1 bit per keystroke pair information can reveal significant information about the content typed. In order to use inter-keystroke timings to infer keystroke sequences, we build a Hidden Markov Model and develop a *n*-Viterbi algorithm for the keystroke sequence inference. To evaluate the effectiveness of the attack, we further build an attacker system, Herbivore, which monitors the network and collects timing information about keystrokes of users' passwords. Herbivore then uses our key sequence prediction algorithm for password prediction. Our experiments show that, for passwords that are chosen uniformly at random with length of 7 to 8 characters, Herbivore can reduce the cost of password cracking by a factor of 50 and hence speed up exhaustive search dramatically. We also propose some countermeasures to mitigate the problem.

We emphasize that the attacks described in this paper are a general issue for any protocol that encrypts interactive traffic. For concreteness, we study primarily SSH, but these issues affect not only SSH 1 and SSH 2, but also any other protocol for encrypting typed data.

[1] In this paper we only consider users who are familiar with keyboard typing and use touch typing.

The outline of this paper is as follows. In Section 2 we discuss in more details about the vulnerabilities of SSH and various simple techniques an attacker can use to learn sensitive information such as the length of users' passwords and the inter-keystroke timings of users' passwords typed. In Section 3 we present our statistical study on users' typing patterns and show that inter-keystroke timings reveal about 1 bit of information per keystroke pair. In Section 4 we describe how we can infer key sequences using a Hidden Markov Model and a *n*-Viterbi algorithm. In Section 5 we describe the design, development and evaluation of an attacker system, Herbivore, which learns users' passwords by monitoring SSH sessions. We propose countermeasures to prevent these attacks in Section 7, and conclude in Section 8.

2 Eavesdropping SSH

The Secure Shell SSH [SSL01, YKS$^+$00b] is used to encrypt the communication link between a local host and a remote machine. Despite the use of strong cryptographic algorithms, SSH still leaks information in two ways:

- First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which leaks the approximate size of the original data.
- Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`). Because the time it takes the operating system to send out the packet after the key press is in general negligible comparing to the inter-keystroke timing (as we have verified), this also enables an eavesdropper to learn the precise inter-keystroke timings of users' typing from the arrival times of packets.

The first weakness poses some obvious security risks. For example, when one logs into a remote site *R* in SSH, all the characters of the initial login password are batched up, padded to an eight-byte boundary if a block cipher is in use, encrypted, and transmitted to *R*. Due to the way padding is done, an eavesdropper can learn one bit of information on the initial login password, namely, whether it is at least 7 characters long or not. The second weakness can lead to some potential anonymity risks since, as many researchers have found previously, inter-keystroke timings can reveal the iden-

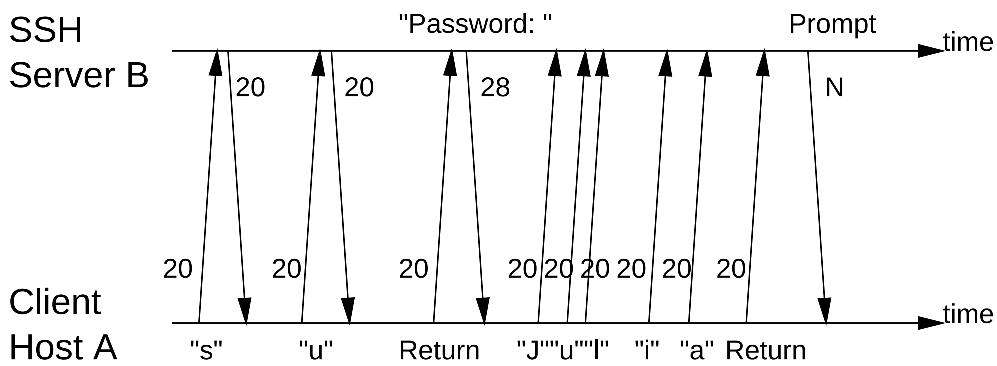


Figure 1: The traffic signature associated with running SU in a SSH session. The numbers in the figure are the size (in bytes) of the corresponding packet payloads.

tity of the user [GLPS80, UW85, LW88, LWU89, JG90, BSH90, MR97, RLCM98, MRW99].

In this section, we show that several simple and practical attacks exploiting these two weaknesses. In particular, an attacker can identify which transmitted packets correspond to keystrokes of sensitive data such as passwords in a SSH session. Using this information, the attacker can easily find out the exact length of users' passwords and even the precise inter-keystroke timings of the typed passwords. Learning the exact length of users' passwords allows eavesdroppers to target users with short passwords. Learning the inter-keystroke timing information of the typed passwords allows eavesdroppers to infer the content of the passwords as we will show in Section 3 and 4.

Traffic Signature Attack We can often exploit properties of applications to identify which packets correspond to the typing of a password. Consider, for instance, the SU command. Assume the user has already established a SSH connection from local host *A* to remote host *B*. When the user types the command SU in the established SSH connection $A \leftrightarrow B$, we obtain a peculiar traffic signature as shown in Figure 1. If the SSH session uses SSH 1.x[2] and a block cipher such as DES for the encryption [NBS77, NIS99], as is common, then the local host *A* sends three 20-byte packets: "s", "u", "Return". The remote host *B* echoes the "s" and "u" in two 20-byte packets and sends a 28-byte packet for the "Password: " prompt. Then *A* sends 20-byte packets, one for each of the password characters, without receiving any echo data packets. *B* then sends some final packets containing the root prompt if SU succeeds, otherwise some failure messages. Thus by checking the traffic against this "su" signature, the attacker can identify when the user issues the SU command and hence learn which packets correspond to the password keystrokes. Note that similar techniques can be used to identify when users type passwords to authenticate to other applications such as PGP [Zim95] in a SSH session.

Multi-User Attack Even more powerful attacks exist when the attacker also has an account on the remote machine where the user is logging into through SSH. For example, the process status command ps can list all the processes running on a system. This allows the attacker to observe each command that any user is running. Again, if the user is running any command that requires a password input (such as su or pgp) the attacker can identify the packets corresponding to the password keystrokes.

Nested SSH Attack Assume the user has already established a SSH session between the local host *A* and remote host *B*. Then the user wants to open another SSH session from *B* to another remote host *C* as shown in Figure 2. In this case, the user's password for *C* is transmitted, one keystroke at a time, across the SSH-encrypted link $A \leftrightarrow B$ from the user to *B*, even though the SSH client on machine *B* patiently waits for all characters of the password before it sends them all in one packet to host C for authentication (as designed in the SSH protocol [YKS$^+$00a]). It is easy to identify such a nested SSH connection using techniques developed by Zhang and Paxson [ZP00b, ZP00a]. Hence in this case the eavesdropper can easily identify the packets corresponding to the user's password on link $A \leftrightarrow B$, and from this learn the length and the inter-keystroke timings of the users' password on host *C*.

[2] The attack also works when ssh 2.x is in use. Only the packet sizes are slightly different.

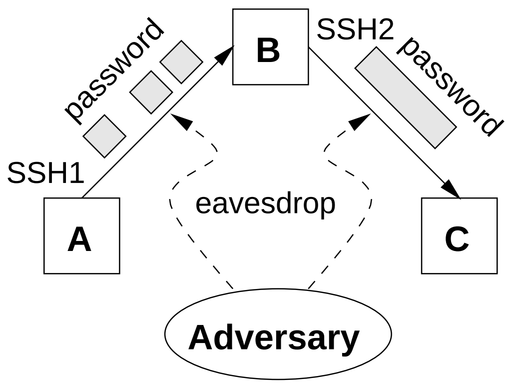


Figure 2: The nested SSH attack.

3 Statistical Analysis of Inter-keystroke Timings

As a first study towards inferring key sequences from timing information, we develop techniques for statistical analysis of the inter-keystroke timings. In this section, we first describe how we collect training data and show some simple timing characteristics of character pairs. We then show how we model the inter-keystroke timing of a given character pair as a Gaussian distribution. We then describe how to estimate quantitatively the amount of information about the character pair that one can learn using the inter-keystroke timing information. Denote the set of character pairs of interest as Q, and let $|Q|$ denote the cardinality of the set Q.

3.1 Data Collection

The two keystrokes of a pair of characters (k_a, k_b) generates four events: the press of k_a, the release of k_a, the press of k_b, and the release of k_b. However, because only key presses (not key releases) trigger packet transmission, an eavesdropper can only learn timing information about the key-press events. Since the main focus of our study is in the scenario where an adversary learns timing information on keystrokes by simply monitoring the network, we focus only on key-press events. The time difference between two key presses is called the *latency* between the two keystrokes. We also use the term *inter-keystroke timing* to refer to the latency between two keystrokes.

In order to characterize how much information is leaked by inter-keystroke timings, we have performed a number of empirical tests to measure the typing patterns of real users. Because passwords are probably the most sensitive data that a user will ever type, we focus only on information revealed about passwords (rather than other forms of interactive traffic).

Our focus on passwords creates many challenges. Passwords are entered very differently from other text: passwords are typed frequently enough that, for many users, the keystroke pattern is memorized and often typed almost without conscious thought. Furthermore, well-chosen passwords should be random and have little or no structure (for instance, they should not be based on dictionary words). As a consequence, naive measurements of keystroke timings will not be representative of how users type passwords unless great care is taken in the design of the experimental methodology.

Our experimental methodology is carefully designed to address these issues. Due to security and privacy considerations, we chose not to gather data on real passwords; therefore, we have chosen a data collection procedure intended to mimic how users type real passwords. A conservative method is to pick a random password for the user (where each character of the password is chosen uniformly at random from a set of 10 letter keys and 5 number keys, independently of all other characters in the password), have the user practice typing this password many times without collecting any measurements, and then measure inter-keystroke timing information on this password once the user has had a chance to practice it at length.

However, we found that, when the goal is to try to identify potentially relevant timing properties (rather than verify conjectured properties), this conservative approach is inefficient. In particular, users typically type passwords in groups of 3–4 characters, with fairly long pauses between each group. This distorts the digraph statistics for the pair of characters that spans the group boundary and artificially inflates the variance of our measurements. As a result we would need to collect a great deal of data for many random passwords before this effect would average out. In addition, it takes quite a while for users to become familiar with long random passwords. This makes the conservative approach a rather blunt tool for understanding inter-keystroke statistics.

Fortunately, there is a less costly way to gather inter-keystroke timing statistics: we gather training data on each pair of characters (k_a, k_b) as typed in isolation. We pick a character pair and ask the user to type this pair 30–40 times, returning to the home row each time between repetitions. For each user, we repeat this for many possible pairs (142 pairs, in our experiments) and we gather data on inter-keystroke timings for each such pair. We collected the latency of each character pair measurement and computed the mean value and the standard deviation. In our experience, this gives better results.

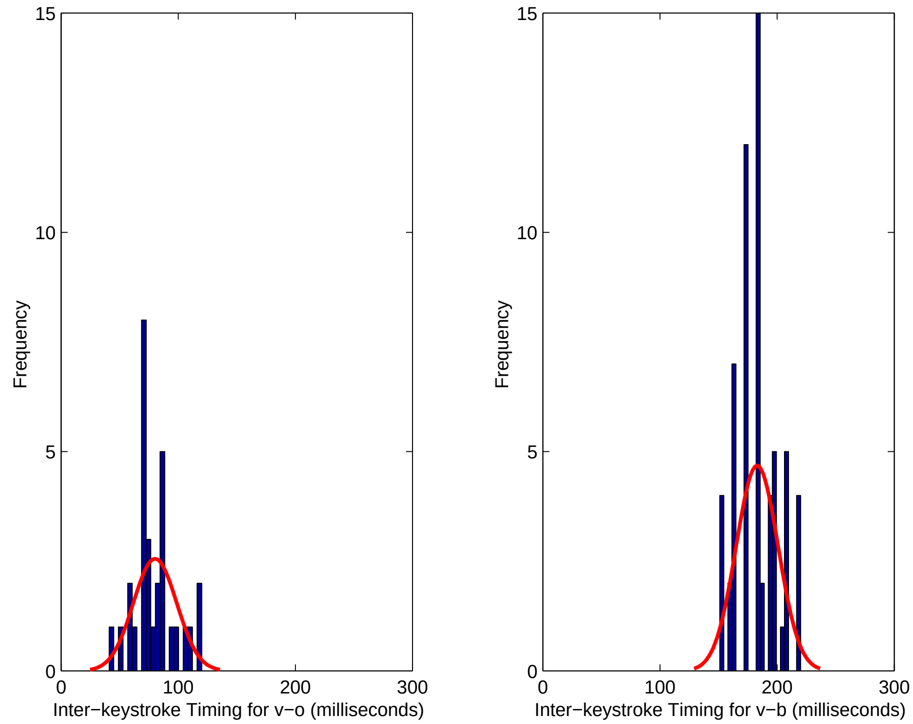


Figure 3: The distribution of inter-keystroke timings for two sample character pairs.

As an example, Figure 3 shows the latency histogram of two sample character pairs. The left model corresponds to the latency between the pair (v, o), and the right model corresponds to (v, b). We can see that the latency between (v, o) is clearly shorter than the latency between (v, b), and the latency distributions of these two sample character pairs are almost entirely non-overlapping.

The optimized data collection approach gives us a more efficient way to study fine-grained details of inter-keystroke statistics without requiring collecting an enormous amount of data. We used data collected in this way to quickly identify plausible conjectures, develop potential attacks, and to train our attack models. As far as we are aware, collecting data on keystroke pairs in isolation does not seem to bias the data in any obvious way. Nonetheless, we also validate all our results using the conservative measurement method (see Section 5).

3.2 Simple Timing Characteristics

Next, we divide the test character pairs into five categories, based on whether they are typed using the same hand, the same finger, and whether they involve a number key:

- Two letter keys typed with alternating hands, i.e., one with left hand and one with right hand;
- Two characters containing one letter key and one number key typed with alternating hands;
- Two letter keys, both typed with the same hand but with two different fingers;
- Two letter keys typed with the same finger of the same hand;
- Two characters containing one letter key and one number key, both typed with the same hand.

Figure 4 shows the histogram of latency distribution of character pairs for each category. We split the whole latency range into six bins as shown in the *x*-axis. Within each category, we put each character pair into the corresponding bin if its mean latency value is within the range of the bin. Each bar in the histogram of a category represents the ratio of the number of character pairs in the associated bin over the total number of character pairs in the category.[3] We can see that all the character pairs that are typed using two different hands take less than 150 milliseconds, while pairs typed using the same hand and particularly the same finger take substantially longer. Character pairs that alternate between one letter key and one number key, but are typed using the same

[3] Hence the sum of all bars within one category is 1.

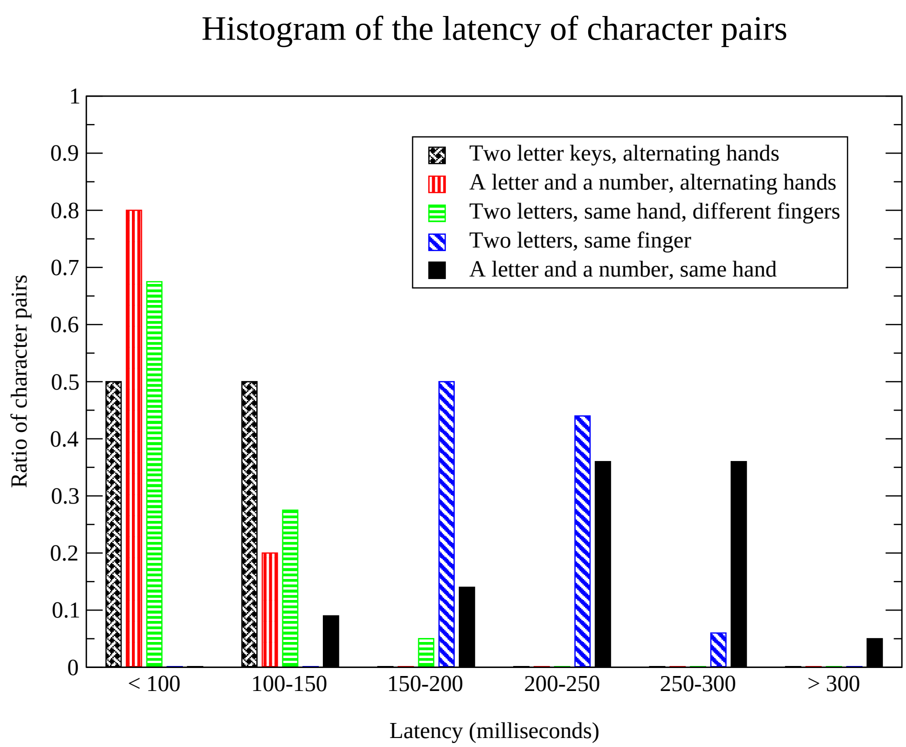


Figure 4: Inter-keystroke timings for character pairs in five different categories. Note that some bars at some positions disappear because the corresponding height is zero.

hand, take the longest time to type. This is simply because two hands offer a certain amount of parallelism, while character pairs typed with one hand require a certain degree of sequential movements and hence tend to take longer. This is especially obvious in the case of one letter and one number pairs typed using one hand. They in general require more hand movement and hence the longest time.[4]

So, if the attacker observes a character pair typed with latency more than 150 milliseconds, he can guess with high probability of success that the character pair is not typed using two different hands and hence can infer about 1 bit of information about the content of the character pair. Because the 142 character pairs are formed from randomly selected letter keys and number keys, they seem likely to form a representative sample of the whole keyboard. Hence this simple classification extends to the whole keyboard, and already indicates that the inter-keystroke timing leaks substantial information about what is typed.

The properties described above are unlikely to be exhaustive. For instance, earlier work on timing attacks on multi-user machines suggested that inter-keystroke timings may additionally reveal which characters in the password are upper-case [Tro98].

[4]Note that here we only consider users that use touch typing.

3.3 Gaussian Modeling

From the plot of the latency distribution of a given character pair, such as the ones shown in Figure 3, we can see that the latency between the two key strokes of a given character pair forms a Gaussian-like unimodal distribution. Hence a natural assumption (which is confirmed by our empirical observations) is that the probability of the latency y between two keystrokes of a character pair $q \in Q$, $\Pr[y|q]$, forms a univariate Gaussian distribution $\mathcal{N}(\mu_q, \sigma_q)$, meaning

$$\Pr[y|q] = \frac{1}{\sqrt{2\pi}\sigma_q} e^{-\frac{(y-\mu_q)^2}{2\sigma_q^2}},$$

where μ_q is the mean value of the latency for character pair q and σ_q is the standard deviation. Given a set of training data $\{(q_i, y_i)\}_{1 \le i \le N}$, where q_i is the i-th character pair and y_i is the corresponding latency in the data collection, we can derive the parameters $\{(\mu_q, \sigma_q)\}_{q \in Q}$ based on *maximum likelihood* estimation, i.e., we compute the mean and the standard deviation for each character pair.

Figure 5 shows the estimated Gaussian models of the latencies of the 142 character pairs. Our empirical result

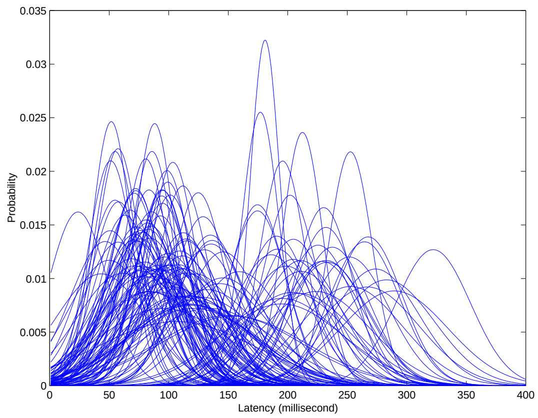
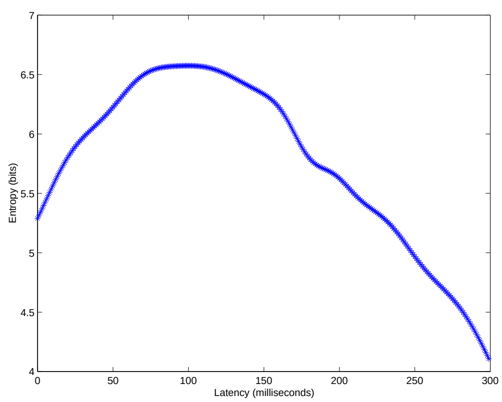
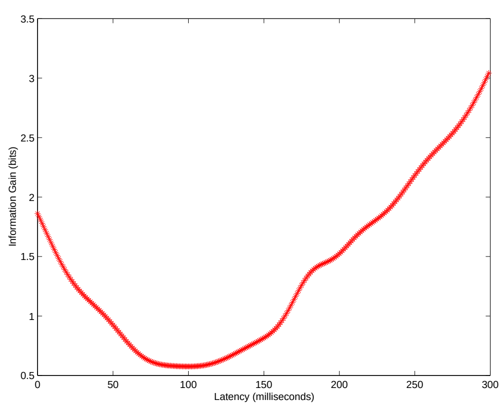


Figure 5: Estimated Gaussian distributions of all 142 character pairs collected from a user.



(a) Entropy of character pairs given a latency observation



(b) Information gain induced by a latency observation

Figure 6: Entropy and information gain as a function of the inter-keystroke latency.

shows that most of the latencies of the character pairs lie between 50 and 250 milliseconds. The average of the standard deviation of the 142 character pairs is about 30 milliseconds. The graph also indicates that the latency distributions of the character pairs severely overlap, which means the inference of character pairs using just latency information is a challenging task.

3.4 Information Gain Estimation

We would like to estimate quantitatively how much information the latency information reveals about the character pairs typed. This will be an upper bound of how much information an attacker can extract from the timing information using any particular method. We estimate it by computing the *information gain* induced by the latency information. If we select a character pair uniformly at random from the character-pair space, and if the attacker does not get any additional information, the entropy of the probability distribution of character pairs to the attacker is $H_0[q] = -\sum_{q\in Q}\Pr[q]\log_2\Pr[q] = \log_2|Q|$. If the attacker learns the latency y_0 between the two keystrokes of the character pair, the estimated entropy of the probability distribution of character pairs to the attacker is $H_1[q|y = y_0] = -\sum_{q\in Q}\Pr[q|y_0]\log_2\Pr[q|y_0]$, where $\Pr[q|y_0] = \frac{\Pr[y_0|q]\cdot\Pr[q]}{\sum_{q\in Q}\Pr[y_0|q]\cdot\Pr[q]}$, and $\Pr[y_0|q]$ is computed using the Gaussian distribution obtained in the parameter estimation phase in the previous subsection. The information gain induced by the observation of latency y_0 is the difference between the two entropies, $H_0[q] - H_1[q|y = y_0]$. Using the parameter estimation of the 142 character pairs obtained in the previous section, we can compute $H_1[q|y = y_0]$ and $H_0[q] - H_1[q|y = y_0]$ as shown in Figure 6(a) and Figure 6(b).

The estimated information gain, also called *mutual information*, is $\mathrm{I}[q;y] = H_0[q] - H_1[q|y] = H_0[q] - \int\Pr[y_0]\cdot H_1[q|y = y_0]\,dy_0$, where $\Pr[y_0] = \sum_{q\in Q}\Pr[y_0|q]\Pr[q]$. From the numerical computation we obtain $\mathrm{I}[q;y] = 1.2$. This means the estimated information gain available from latency information is about 1.2 bits per character pair when the character pair has uniform distribution. Hence the attacker could potentially extract 1.2 bits of information per character pair by using the latency information in this case. Because the character pairs in our experiments are selected uniformly at random from all letter and number keys, we expect that they will be representative of the whole keyboard. Intuitively, Figure 5 is a sufficiently-large random sampling of a much denser graph containing the latency distributions of all possible character pairs. More detailed analysis shows that the estimated information gain computed using 142 sample character pairs is a good estimate of the information gain when the character-pair space includes all letter and number character pairs. This estimate is comparable to the back-of-the-envelope calculation in Section 3.2 based on our classification into five categories of keystroke pairs.

Because the entropy of written English is so low (about 0.6–1.3 bits per character [Sha50]), the 1.2-bit information gain per character pair leaked through the latency information seems to be significant. [5] For example, we can expect that users' PGP passphrases will often contain only 1 bit of entropy per character. Hence the latency information may reveal significant information about PGP passphrases.

The information gain curve in Figure 6(b) shows a convex shape. Note that latencies greater than 175 milliseconds are relatively rare; however, whenever we see such a long time between keystrokes, we learn a lot of information about what was typed, because there are not many possibilities that would lead to such a large latency. The character pairs that take longer than 175 milliseconds to type are mostly pairs containing number keys or pairs typed with one finger. Hence this analysis suggests that passwords containing number keys or character pairs that are typed with one finger are particularly vulnerable to such timing attacks.

Another interesting observation is that the mean of the standard deviations of the character pairs is only about 30 milliseconds as shown in our experiments, while the standard deviation of round-trip time on the Internet in many cases is less than 10 milliseconds [Bel93]. Therefore even when the attacker is far from the SSH client host, he can still get sufficiently-precise inter-keystroke timing information. This makes the timing attack even more severe.

4 Inferring Character Sequences From Inter-Keystroke Timing Information

In this section, we describe how we can infer character sequences using the latency information. In particular, we model the relationship of latencies and character sequences as a Hidden Markov Model [RN95]. We extend the standard Viterbi algorithm to an n-Viterbi algorithm that outputs the n most likely candidate character sequences. We further estimate how many bits of information about the real character sequence this algorithm extracts from the latency information and show it is nearly optimal.

4.1 Hidden Markov Model

In general, a Markov Model is a way of describing a finite-state stochastic process with the property that the probability of transitioning from the current state to another state depends only on the current state, not on any prior state of the process [RN95]. In a Hidden Markov Model (HMM), the current state of the process cannot be directly observed. Instead, some outputs from the state are observed, and the probability distribution of possible outputs given the state is dependent only on the state. Using a HMM, one can infer information about the prior path the process has taken from the sequence of observed outputs of the states, and efficient algorithms are known for working with HMM's. Because of this, HMM's have been widely used in areas such as speech recognition and text modeling.

In our setting, we consider each character pair of interest as a hidden (non-observable) state, and the latency between the two keystrokes of the character pair as the output observation from the character-pair state. Each state corresponds to a pair of characters, so that the typing of a character sequence $K_0, \ldots, K_T$, is a process that goes through T states, $q_1, \ldots, q_T$, where $q_t (1 \leq t \leq T)$ represents the t-th character pair (K_{t-1}, K_t) typed. Let $y_t (1 \leq t \leq T)$ denote the observed latency of state q_t. Then we model the typing of a character sequence as a HMM. This means we make two assumptions. First, the probability of transition from the current state to another state is only dependent on the current state, not on the prior path of the process. If the character sequence is a password chosen uniformly at random, this assumption obviously holds. In the case of text, this assumption does not hold strictly but experience in speech recognition and text modeling shows that some extensions to HMM still work well [RN95]. Second, the probability distribution of the latency observation is only dependent on the current character pair and not on any previous characters in the sequence. This assumption might hold for some cases and not for other cases where the typing of previous characters changes the position of the hand and influences the typing of later character pairs. However, making this assumption makes our analysis and inference algorithm much simpler and still gives good results as shown from the experiments. Hence, we use a HMM to model the typing of character sequences as shown in Figure 7.

As in the previous section, we assume the set of possible character pairs is Q, hence the set of possible states in the HMM is Q. We assume that the probability of

[5] Note that the 1.2-bit information gain is estimated for the case of randomly selected passwords where the sequence of characters have a uniform distribution. However, this is not the case for texts. More careful calculation is needed to estimate the information gain in the case of natual text.

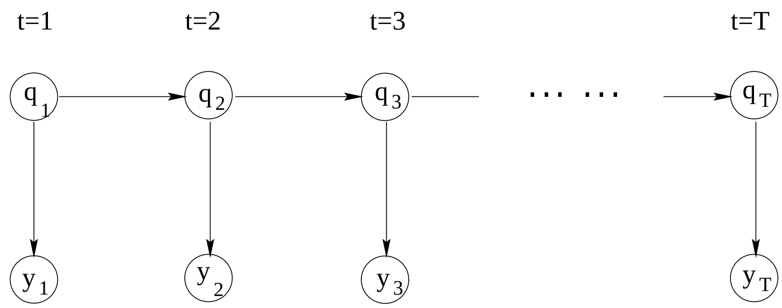


Figure 7: A representation of a trace of a HMM. Each vertical slice represents a time step. In each time slice, the top node q_t is a variable representing a character pair, and the bottom node y_t is the observable variable denoting the latency between the two keystrokes.

the latency y of a character pair q, $\Pr[y|q]$ ($q \in Q$), is a Gaussian distribution $\mathcal{N}(\mu_q, \sigma_q)$, where the parameters $\{(\mu_q, \sigma_q)\}_{q \in Q}$ are obtained using the maximum likelihood estimation.

4.2 The *n*-Viterbi Algorithm for Character Sequence Inference

Given an observation $\vec{y} = (y_1, y_2, \ldots, y_T)$, a sequence of latencies of some character sequence from a user's typing, we would like to infer the real character sequence that the user has typed. For each possible character sequence $\vec{q} = (q_1, q_2, \ldots, q_T)$, we can compute how likely the character sequence is given the observation, namely $\Pr[\vec{q}\,|\vec{y}]$. The probability $\Pr[\vec{q}\,|\vec{y}]$ essentially gives a ranking for the candidate character sequence $\vec{q}$: the higher $\Pr[\vec{q}\,|\vec{y}]$ is, the more likely $\vec{q}$ is the real character sequence. We use $\vec{q}^{\,*}$ to denote the most-likely sequence, which is the sequence that corresponds to the highest value of $\Pr[\vec{q}|\vec{y}]$ for all possible $\vec{q}$ with regard to a given $\vec{y}$.

The Viterbi algorithm is widely used in solving the most likely sequence of states given a sequence of observation in HMM problems [RN95]. An naive way of computing $\vec{q}^{\,*}$ would compute $\Pr[\vec{q}|\vec{y}]$ for all possible $\vec{q}$, and hence requires $O(|Q|^T)$ running time. The Viterbi algorithm uses dynamic programming for a running time complexity $O(|Q|^2 T)$.

In our setting, because the latency distributions of different character pairs highly overlap, the probability that the most likely sequence is the right sequence will be very low. Hence, instead of just computing the most likely sequence, we need to compute the n most likely sequences and hope the real sequence will be in the n most likely sequences with high probability for n greater than a certain threshold. Hence we extend the standard Viterbi algorithm to n-Viterbi algorithm to output the n most-likely sequences with running time complexity $O(n|Q|^2 T)$. We give a detailed description of the n-Viterbi algorithm in Appendix A.

4.3 How to Estimate the Effectiveness of the *n*-Viterbi Algorithm

We would like to estimate how big the threshold n has to be such that the real character sequence will be among the n most-likely sequences with sufficiently high probability. In an experiment if the real character sequence appears in the n most-likely sequences, we say the experiment is a success with regard to the threshold n, otherwise, a failure. The probability of such defined success is a function of n. It is easy to see that the function is monotonically increasing with regard to n. If for a small n, the success probability is already high, this means the algorithm is very effective because it filters out most of the sequences and hence one only needs to try a small set of candidates before finding the real sequence. On the other hand, if we need a high threshold of n to get a sufficiently high success probability, then the algorithm is less effective: one would need to try many more candidates before finding the real sequence. Note that from Section 3.4 we see that the timing information reveals about 1.2 bits of information per character pair. For the case of a random password of length $T+1$, which forms T consecutive character pairs, the latency information could reveal approximately $1.2T$ bits of information about the real password sequence. Hence this is an upper bound on the effectiveness of the algorithm to infer character sequences using latency information. We would like to estimate how close our algorithm is compared to the upper bound.

First, we look at the simple case when $T = 1$. Given a latency observation y of a character pair q, we compute the probability $\Pr[q'|y], q' \in Q$, and select the n most-likely character pairs $\Phi = \{q_{j_1}, \ldots, q_{j_n}\}$. We would like to compute the probability that the real character pair q is in the set Φ over all possible values of y. To simplify

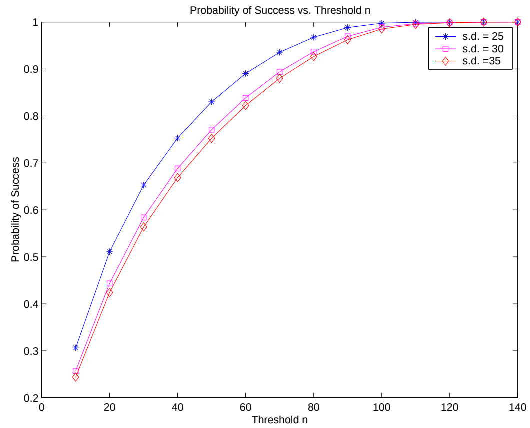


Figure 8: The probability that the n-Viterbi algorithm outputs the correct password before the first n guesses, graphed as a function of n.

the numerical computation, we approximate the result by assuming that all the Gaussian distributions have the same standard deviation σ. This is a good approximation of the real experiment: as we see in the Figure 5, most keypairs have a standard deviation between 25–35 milliseconds.

Figure 8 graphs the probability that the real character pair appears within the n most-likely character pairs against the threshold n. The top curve is when $\sigma = 25$, the middle curve is when $\sigma = 30$, and the bottom curve is when $\sigma = 35$. Using the middle curve, we get that when $n = 70$ the probability of success is 90%, meaning that with 90% probability, the real character pair appears in the 70 most-likely sequences output by the n-Viterbi algorithm. Let's denote such a threshold corresponding to the 90% success probability as n^*. Thus $\log_2(|Q|/n^*) = 1$ is the approximate number of bits of information per character pair the algorithm extracts. Note that from the previous section we see that the latency information reveals about 1.2 bits of information per character pair. Hence our n-Viterbi algorithm is near-optimal.

In the case of uniformly randomly chosen passwords of length $T+1$, the number of bits of information the algorithm can extract is approximately $T \cdot \log_2(|Q|/n^*) \approx T$, which is close to the optimal value $1.2T$ bits.

5 Building Herbivore and Timing Attacks on SSH

To evaluate the effectiveness of our timing attacks to SSH, we build an attacker program that we call *Herbivore.* In this section, we describe the experiment results

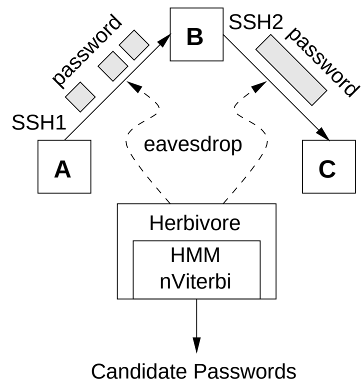


Figure 9: The Herbivore architecture.

of using Herbivore to learn users' passwords.

5.1 Herbivore Preying for Passwords

We built an attacker engine Herbivore as shown in Figure 9. It monitors the network and collects the arrival times of packets. Using the technique described in Section 2, Herbivore infers which packets correspond to the user's SSH passwords when the user opens an SSH session to another host within an established SSH connection. Herbivore then measures the inter-arrival times between packets containing the password characters and uses our n-Viterbi algorithm to generate a list of candidate passwords. The candidate passwords are sorted in decreasing order of the probability $\Pr[\vec{q}\ |y]$, and in our experiments we record the position of the real password in the candidate list. We report the position of the password as a percentage, so with m possible passwords in total, if the real password appears at position u in the ordered candidate list, we say the real password appears at the top $\frac{100u}{m}$%. This gives a natural way to quantify the effectiveness of our approach.

5.2 Optimization for Long Character Sequences

The complexity of the n-Viterbi algorithm is linear in the number n of candidates it outputs. As the length of the password grows, the space of possible passwords grows exponentially. If the n-Viterbi algorithm can only rule out a constant fraction of the password space, n would also grow exponentially as the password length grows. Hence the algorithm might be inefficient when the password is long. In particular, we observed that memory usage can grow substantially for longer passwords.

Also, and more importantly, we observed in the experiments that users tend to type long passwords in segments of 3 to 5 letters and pause between the segments. If we

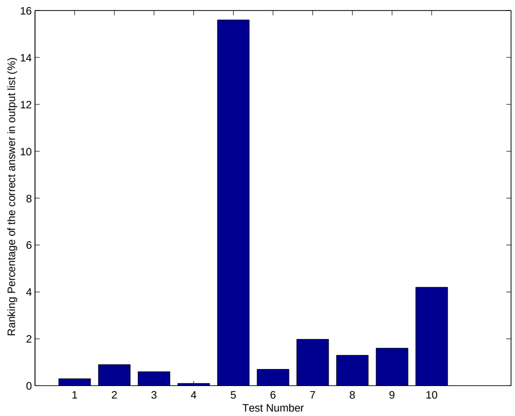


Figure 10: The percentage of the password space tried by Herbivore in 10 tests before finding the right password.

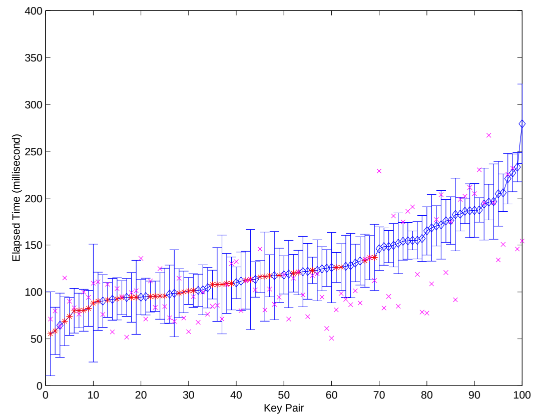


Figure 11: A comparison of two users' typing patterns. The "diamond" symbols show the mean values of the latencies of one user, with an error-bar indicating one standard deviation. The "x" symbol indicates the mean values of the latencies of another user.

use the timing between the segments for the prediction, it might bias our predictions since typically such pauses are noticeably longer than most other inter-keystroke latencies. Fortunately, this large difference means that pauses between groups of password characters can be clearly identified before we apply the *n*-Viterbi algorithm.

Hence to reduce the bias and to reduce the memory requirements of the algorithm, we break the timing information of the password into segments containing 3 or 4 latency intervals. We use each segment to form a HMM and then at the end combine the result from different segments to form the candidate password ordering.

5.3 Experimental Results for Password Inference for a Single User

We measure the effectiveness of our *n*-Viterbi algorithm at cracking passwords through empirical measurements. In our experiment, we use training data compiled from isolated keypairs to train the HMM. Then, we pick a random password for the user. We have the user use this password to authenticate to another SSH session within an established SSH session as shown in Figure 9, and we apply our *n*-Viterbi algorithm to simulate an attack on this password. Note that we have the test user type the password many times before the test to ensure familiarity with the password, and we try to deduce the user's password using training data from the same user.

All passwords are selected uniformly at random from the character space as in the experiment in Section 3, so they contain no structure. Recovering such passwords is the hardest case for the attacker, so if timing analysis can recover information in such a scenario, we can expect that timing analysis will be an even greater threat in settings where passwords are chosen less carefully.

We performed tests for 10 different passwords, each of length 8. Figure 10 shows the percentage of the positions of the real password in the ordered candidate lists output by the *n*-Viterbi algorithm. For example, 0.3% means that the real password appeared at the top 0.3% position in the output candidate list. These experiments indicate that on average the real password is located within the top 2.7% of the candidate ranking list. The median position is about 1%, so about half the time the password will be in the top 1% of the list of candidates produced by our *n*-Viterbi algorithm. Therefore, in order to crack the password, Herbivore only needs to test $1/50$ times as many passwords as brute-force search, on average.

The $50\times$ reduction in workfactor compared to exhaustive search corresponds to a total of 5.7 bits of information learned per password using the latency information. This is close to the information gain analysis in Sections 3 and 4, which predicted a gain of about 1 bit per keystroke pair: recall that the passwords in this test are of length 8, so each password contains 7 keystroke pairs. We attribute the difference to minor variation between the distributions of inter-keystroke timings in random passwords and the distribution of timings for character pairs typed in isolation.

For ease of testing, our experiments were on passwords with a reduced set of possible characters. However, we can expect these results to carry over to passwords chosen from the full set of possible characters. Assuming that the information gain available from inter-keystroke timing information is about 1 bit per character pair even

Training Set	Test Set	Test Cases				
		Password 1	Password 2	Password 3	Password 4	Password 5
User 1	User 1	15.6%	0.7%	2.0%	1.3%	1.6%
User 1	User 2	62.3%	15.2%	7.0%	14.8%	0.3%
User 1	User 3	6.4%	N/A	1.8%	3.1%	4.2%
User 1	User 4	1.9%	31.4%	1.1%	0.1%	28.8%
User 2	User 1	4.9%	1.3%	1.6%	12.3%	3.1%
User 2	User 2	30.8%	15.0%	2.8%	3.7%	2.9%
User 2	User 3	4.7%	N/A	5.3%	6.7%	38.4%
User 2	User 4	0.7%	16.8%	3.9%	0.6%	5.4%

Table 1: Success rates for password inference with multiple users. The numbers are the percentage of the search space the attacker has to search before he finds the right password.

when we extend to the whole keyboard, we expect to see this 50 times reduction in workfactor for passwords of length 7–8 even when the passwords are chosen randomly from all letter and number keys. This 50× reduction can make password cracking more practical. For example, for a password containing randomly-selected lower-case letter keys and number keys, without timing information, the attacker would need to try $36^8/2$ candidate passwords on average before he finds the right one. Benchmarks indicate that a 840 MHz Pentium III can check about $250,000$ candidate passwords per second in a off-line dictionary attack. Thus, exhaustive search would take about 65 PC-days to crack a password composed of randomly-selected lower-case letter keys and number keys. If the attacker uses the timing information, the computation can be done in 1.3 days, which makes the crack 50× more practical.

5.4 Experimental Results for Password Inference for Multiple Users

One potential weakness in our simulations is that real-world attackers might not be able to get as much training data from the victim for the statistical analysis as we had available in our experiments. However, we argue next that this is unlikely to pose an effective defense against timing attacks: there are other ways that attackers can obtain the training data required for the attack.

One simple observation is that the attacker can easily get his own typing statistics, or the typing statistics of a co-conspirator. Hence it is important to evaluate how well the password inference techniques perform when using one person's typing statistics to infer passwords typed by another person.

In this experiment, we collected the typing statistics of two users, User 1 and User 2. An interesting result is that 75% of the character pairs take about the same latency to type for both two users: in other words, the difference between the average latencies of the two users for such character pairs is smaller than one standard deviation. Similarly, the simple timing characteristics reported in Section 3.2—e.g., keypairs typed with alternate pairs tend to have much lower inter-keystroke latency than keypairs typed with the same hand—were observed to be essentially user-independent. This suggests that typing statistics have a large component that is common across a broad user population and which thus can be exploited by attackers even in the absence of any training data from the victim.

To test this hypothesis further, we had four users (including User 1 and 2, from our previous experiments) type the same set of five randomly-selected passwords. Passwords 1 and 2 have length 8. Passwords 3 and 4 have length 7, and password 5 has length 6. Herbivore then runs the n-Viterbi algorithm using the typing statistics from User 1 and 2 to infer passwords typed by the four test users separately. Table 1 shows the percentage position of the real passwords occurred in the output candidate ranking list, which is the percentage of the password space the attacker has to search before he finds the right password. User 3 did not type Password 2 so the entry is not available.

This experiment shows several interesting results:

- Unsurprisingly, inferring a user's password can in general be done somewhat more effectively if one uses training data from the same user rather than training data from other users.
- The distance between the typing statistics of two users can vary significantly according to how one chooses the pair of users. A user U_a's typing pattern might be more similar to user U_b's than to user U_c's. Thus it can give better results to use U_b's training data than U_c's training data to infer passwords typed by U_a. In this experiment, it shows

that in general using User 1's training data gives a better result to infer passwords typed by User 3 than using User 2's training data. And User 2's training data gives a better inference for passwords typed by User 4 than User 1's training data.

- Most importantly, this experiment shows that training data from one user can be successfully applied to infer passwords typed by another user. Hence the attack can be effective even when the attacker does not have typing statistics from the victim.

5.5 Extensions

We expect that Herbivore could also be used to infer information about text or commands that users type. The entropy of written English is very low (about 0.6–1.3 bits per character [Sha50]) in comparison to the amount of information leaked by inter-keystroke timings (about 1 bit of information per key pair; see Section 3). However, mounting such an attack would appear to require better models of written text [RN95]. In any case, we have not studied such a scenario in our experiments, and we leave this for future work.

6 Related Work

Timing analysis has previously been used by Kocher to attack cryptosystems [Koc95]. Trostle exploited a similar idea, showing how a malicious user on a multi-user workstation can gain information about other users' passwords using CPU timings [Tro98]. We expect our Hidden Markov Model techniques might find applications in Trostle's threat model as well.

Most recently, other researchers have independently pointed out the possibility of timing attacks on `SSH` [DS01]. Some of their observations reveal additional weaknesses in `SSH`: For instance, they noted that the `SSH 1.x` protocol reveals the exact length of passwords, because ciphertexts contain a length field sent in the clear (`SSH 2` does not have this problem); they discussed how to deal with the presence of backspace characters; and, they initiated an investigation of the impact of timing attacks on other session data (such as shell commands typed in the `SSH` session).

7 Countermeasures

Although `SSH` provides an encrypted and authenticated link between the local host and the remote machine, an eavesdropper can still learn information about typed keystrokes due to two weaknesses of `SSH`. First, every individual keystroke that a user types is sent to the remote machine in an individual IP packet (except for meta keys such as `Shift` and `Ctrl`); second, as soon as command output is available on the remote machine, it is sent to the local host in one or multiple IP packets, leaking information on the approximate size of the output. We have shown in this paper how these seemingly minor weaknesses lead to severe real-world attacks.

Note that in our traffic signature attack, the attacker can tell that the user is typing passwords because there are no echo packets. So one way to fix this problem is that when the server detects that the echo mode is turned off, the server can return dummy packets that will be ignored by the client when it receives keystroke packets from the client. This fix can reduce the effectiveness of the traffic signature attack but could fail in other attacks such as our nested `SSH` attack where the attacker can guess when the user is typing his password by simply monitoring the network connections. This fix does not prevent inter-keystroke timing information, though.

To prevent the attacks, we need to prevent the leakage of the timing information of the keystrokes. One naive approach might be to modify SSH so that upon receiving a keystroke with latency less than η milliseconds from the previous keystroke, the program will delay the packet by a random amount of up to η milliseconds. Because our experiment indicates that the spectrum of the latency between two keystrokes of continuous typing is between 0–500 milliseconds, we could set $\eta = 500$ for example, and such a random delay would randomize the timing information of the keystrokes. Such a random delay imposes an overhead of about 250 milliseconds on average. Unfortunately, if the attacker can monitor the same user login many times and compute the average of the latencies of the password sequences, he can reduce the effectiveness of the randomized noise. For example, if the attacker can get the timing information of a user's `SSH` authentication for 50 times, the noise contributed by the random delay is only about 20–40 milliseconds. So we should not use this method.

A better way to prevent leakage of inter-keystroke timing information is to send traffic at a constant rate of λ packets per second when the link is active. Choosing λ presents a tradeoff between usability and overhead: Increasing λ reduces the dummy traffic but cause longer latency for the user. Assume, for example, that we set $\lambda = 50$ milliseconds. Since the latency between two keystrokes is usually greater than 50 milliseconds and the network delay is already at least in the tens of milliseconds, this may be a reasonable tradeoff between communication overhead and additional delay. In such a scenario, the `SSH` client would always send a data packet every 50 milliseconds. Assuming 64 byte packets (40 bytes for IP and TCP headers, and 24 bytes for SSH data), the communication overhead is 1280 bytes/second, which can even fit in low-bandwidth connections, such as modem connections. If no real data needs to be sent, the client will send dummy traffic which the remote machine ignores.[6] If the user types multiple keys in a single time period, the keystrokes are buffered and sent together in the next scheduled packet. While this method prevents the eavesdropper from learning timing information about keystrokes typed at the client side, it does not prevent information leakage from the size of response packets from the remote machine. Hence the server side would also need to send response traffic at a constant packet rate similar to the client side.

8 Conclusion

In this paper, we identified several serious security risks in `SSH` due to two weaknesses of `SSH`: First, the transmitted packets are padded only to an eight-byte boundary (if a block cipher is in use), which reveals the approximate size of the original data. Second, in interactive mode, every individual keystroke that a user types is sent to the remote machine in a separate IP packet immediately after the key is pressed (except for some meta keys such `Shift` or `Ctrl`), which leaks the inter-keystroke timings of users' typing. We showed that these two weaknesses reveal a surprising amount of information on passwords and other text typed over `SSH` sessions (about 1 bit of information per character pair in the case of randomly chosen passwords). This suggests that `SSH` is not as secure as commonly believed.

The lessons we learned and the techniques we developed in this paper apply to a general class of protocols that aim to provide secure channels between machines. We show that timing information opens a new set of risks, and we recommend that developers take care when designing these types of protocols.

Acknowledgement

We would like to thank Adrian Perrig for his great help through all phases of the project. We are indebted to Kris Hildrum, Doantam Phan and Robert Johnson for their help in the testing phase. We would also like to thank Eric Xing for discussions on statistical techniques. Finally we would like to thank Nikita Borisov, Monica Chew, Kris Hildrum, Robert Johnson, and Solar Designer for their helpful comments on the paper.

[6] If after a certain timeout (e.g., 10λ) there is still no real data to send, the client can consider the current link is inactive and stop sending dummy traffic until it has data to send again. The timeout period provides a tradeoff between security and efficiency.

References

[Bel93] Steven M. Bellovin. Packets found on an internet. *Computer Communications Review*, 23(3):26–31, July 1993.

[BSH90] S. Bleha, C. Slivinksy, and B. Hussein. Computer-access security systems using keystrokes dynamics. In *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-12*, volume 12, December 1990.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security – Repelling the Wily Hacker*. Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63357-4.

[DS01] Solar Designer and Dug Song. Passive analysis of SSH (secure shell) traffic. Openwall advisory OW-003, March 2001.

[GLPS80] R. Gaines, W. Lisowski, S. Press, and N. Shapiro. Authentication by keystroke timing: Some preliminary results. Technical Report Rand report R-256-NSF, Rand corporation, 1980.

[GS96] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, 1996.

[JG90] Rick Joyce and Gopal Gupta. Identity authentication based on keystroke latencies. *Communications of the ACM*, 33(2):168 – 176, February 1990.

[Koc95] P. Kocher. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other cryptosystems using timing attacks. In *Advances in cryptology, CRYPTO '95*, pages 171–183. Springer-Verlag, 1995.

[LW88] G. Leggett and J. Williams. Verifying identity via keystroke characteristics. *International Journal of Man-Machine Studies*, 28(1):67–76, 1988.

[LWU89] G. Leggett, J. Williams, and D. Umphress. Verification of user identity via keystroke characteristics. *Human Factors in Management Information Systems*, 1989.

[MR97] Fabian Monrose and Avi Rubin. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 48–56, April 1997.

[MRW99] F. Monrose, M. K. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, November 1999.

[NBS77] National Bureau of Standards. Specification for the Data Encryption Standard. Federal Information Processing Standards Publication 46 (FIPS PUB 46), January 1977.

[NIS99] U. S. National Institute of Standards and Technology (NIST). Data Encryption Standard (DES). Draft Federal Information Processing Standards Publication 46-3 (FIPS PUB 46-3), January 1999.

[RLCM98] J. A. Robinson, V. M. Liang, J. A. Chambers, and C. L. MacKenzie. Computer user verification using login string keystroke dynamics. *IEEE Transactions on System, Man, and Cybernetics*, 28(2), 1998.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence, A modern approach*. Prentice Hall, 1995.

[Sha50] Claude E. Shannon. Prediction and Entropy of Printed English. Bell Sys. Tech. J (3), 1950.

[SSL01] IETF Secure Shell Working Group (SECSH). `http://www.ietf.org/html.charters/secsh-charter.html`, 2001.

[Tro98] Jonathan Trostle. Timing attacks against trusted path. In *IEEE Symposium on Security and Privacy*, 1998.

[UW85] D. Umphress and J. Williams. Identity verification through keyboard characteristics. *International Journal of Man-Machine Studies*, 23(3):263–273, 1985.

[YKS+00a] T. Ylönen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH authentication protocol. Internet Draft, Internet Engineering Task Force, May 2000. Work in progress.

[YKS+00b] T. Ylönen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft, Internet Engineering Task Force, May 2000. Work in progress.

[Ylö96] Tatu Ylönen. SSH – Secure Login Connections over the Internet. In *Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[Zim95] Philip R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995. ISBN 0-262-74017-6.

[ZP00a] Yin Zhang and Vern Paxson. Detecting backdoors. In *Proc. of 9th USENIX Security Symposium*, August 2000.

[ZP00b] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proc. of 9th USENIX Security Symposium*, August 2000.

A The *n*-Viterbi Algorithm

The Viterbi algorithm is widely used in solving HMM problems. Given an observation $(y_1, \ldots, y_T)$ of a HMM, the Viterbi algorithm inductively computes the most likely sequence $(q_1, q_2, \ldots, q_t)$ that generated the observation for each $t = 1, 2, \ldots, T$. Let $S(q_t)$ be the most likely sequence at time t that ends with state q_t, with corresponding posterior probability $V(q_t)$. The Viterbi algorithm starts with

$$S(q_1) = q_1 \qquad \text{and} \qquad V(q_1) = \Pr[q_1|y_1],$$

and computes

$$V(q_t) = \max_{q_{t-1}} \Pr[y_t|q_t] \Pr[q_t|q_{t-1}] V(q_{t-1})$$

Then we let q_{t-1} be the state that maximizes the above expression and define $S(q_t)$ to be $S(q_{t-1})|q_t$. The final result of the Viterbi algorithm returns the most likely sequence of a given sequence of observations.

We extend the Viterbi algorithm to the *n*-Viterbi algorithm, which returns the *n* most likely sequences given a sequence of observations. Figure 12 shows a diagram of the *n*-Viterbi algorithm. At each time slice t, we associate a list with each possible state node that keeps track of the *n* most likely sequences that lead to the state at that time slice.

Let $S^n(q_t)$ denote the set of the n most likely sequences ending with state q_t at time t, with corresponding posterior probabilities $V^n(q_t)$. At time $t = 1$, we initialize the *n*-Viterbi algorithm in the same way as the Viterbi algorithm,

$$S^n(q_1) = \{q_1\} \qquad \text{and} \qquad V^n(q_1) = \Pr[q_1|y_1].$$

For time t, we let

$$V^n(q_t) = \text{nmax} \quad \{\Pr[y_t|q_t] \Pr[q_t|q_{t-1}] v : q_{t-1} \in Q, v \in V^n(q_{t-1})\}$$

where nmax denotes the set of the n largest values. We let $S^n(q_t)$ be the set n highest-probability sequences corresponding to the choice of $V^n(q_t)$ above.

Except for the first and the second step, at each time slice, for each possible state, we need to go through $n \cdot |Q|$ possibilities and compute the n most likely sequences that lead to that state at that time slice. Hence the complexity of *n*-Viterbi algorithm is $O(n|Q|^2 T)$.

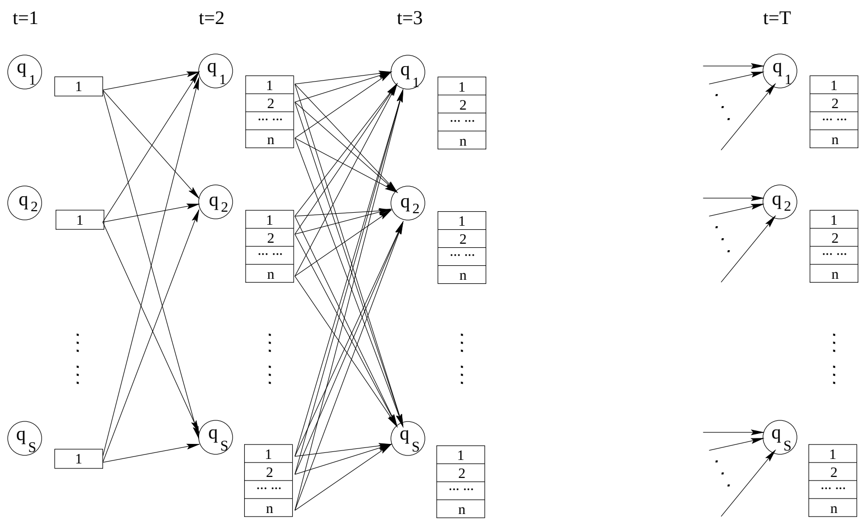


Figure 12: A pictorial representation of the *n*-Viterbi Algorithm. Each vertical slice represents a time step, and each node represents a possible state at a particular time slice. The list associated with each node stores the *n* most likely sequences ending with that state up to that time slice.