Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems

Kehuan Zhang
Indiana University, Bloomington
kehzhang@indiana.edu

XiaoFeng Wang
Indiana University, Bloomington
xw7@indiana.edu

Abstract

A multi-user system usually involves a large amount of information shared among its users. The security implications of such information can never be underestimated. In this paper, we present a new attack that allows a malicious user to eavesdrop on other users' keystrokes using such information. Our attack takes advantage of the stack information of a process disclosed by its virtual file within *procfs*, the process file system supported by Linux. We show that on a multi-core system, the ESP of a process when it is making system calls can be effectively sampled by a "shadow" program that continuously reads the public statistical information of the process. Such a sampling is shown to be reliable even in the presence of multiple users, when the system is under a realistic workload. From the ESP content, a keystroke event can be identified if they trigger system calls. As a result, we can accurately determine inter-keystroke timings and launch a timing attack to infer the characters the victim entered.

We developed techniques for automatically analyzing an application's binary executable to extract the ESP pattern that fingerprints a keystroke event. The occurrences of such a pattern are identified from an ESP trace the shadow program records from the application's runtime to calculate timings. These timings are further analyzed using a Hidden Markov Model and other public information related to the victim on a multi-user system. Our experimental study demonstrates that our attack greatly facilitates password cracking and also works very well on recognizing English words.

1 Introduction

Multi-user operating systems and application software have been in use for decades and are still pervasive today. Those systems allow concurrent access by multiple users so as to facilitate effective sharing of computing resources. Such an approach, however, is fraught with security risks: without proper protection in place, one's sensitive information can be exposed to unintended parties on the same system. This threat is often dealt with by an access control mechanism that confines each user's activities to her compartment. As an example, programs running in a user's account are typically not allowed to touch the data in another account without the permission of the owner of that account. The problem is that different users do need to interact with each other, and they usually expect this to happen in a convenient way. As a result, most multi-user systems tend to trade security and privacy for functionality, letting certain information go across the boundaries between the compartments. For example, the process status command `ps` displays the information of currently-running processes; while this is necessary for the purpose of system administration and collaborative resource sharing, the command also enables one to peek into others' activities such as the programs they run.

In this paper, we show that such seemingly minor information leaks can have more serious consequences than the system designer thought. We present a new attack in which a malicious user can eavesdrop on others' keystrokes using nothing but her non-privileged account. Our attack takes advantage of the information disclosed by procfs [19], the process file system supported by most Unix-like operating systems such as Linux, BSD, Solaris and IBM AIX. Procfs contains a hierarchy of virtual files that describe the current kernel state, including statistical information about the memory of processes and some of their register values. These files are used by the programs like `ps` and `top` to collect system information and can also help software debugging. By default, many of the files are readable for all users of a system, which naturally gives rise to the concern whether their contents could disclose sensitive user information. This concern has been confirmed by our study.

The attack we describe in this paper leverages the

procfs information of a process to infer the keystroke inputs it receives. Such information includes the contents of the extended stack pointer (ESP) and extended instruction pointer (EIP) of the process, which are present in the file `/proc/pid/stat` on a Linux system, where `pid` is the ID of the process. In response to keystrokes, an application could make system calls to act on these inputs, which is characterized by a sequence of ESP/EIP values. Such a sequence can be identified through analyzing the binary executables of the application and used as a pattern to fingerprint the program behavior related to keystrokes. To detect the keystroke event at runtime, we can match the pattern to the ESP/EIP values acquired through continuously reading from the `stat` file of the application's process. As we found in our research, this is completely realistic on a multi-core system, where the program logging those register values can run side by side with its target process. As such, we can figure out when a user strokes a key and use inter-keystroke timings to infer the key sequences [26]. This attack can be automated using the techniques for automatic program analysis [20, 23].

Compared with existing side-channel attacks on keystroke inputs [26, 3], our approach significantly lowers the bar for launching a successful attack on a multi-user system. Specifically, attacks using keyboard acoustic emanations [3, 33, 2] require physically implanting a recording device to record the sound when a user's typing, whereas our attack just needs a normal user account for running a non-privileged program. The timing attack on SSH proposed in the prior work [26] estimates inter-keystroke timings from the packets transmitting passwords. However, these packets cannot be deterministically identified from an encrypted connection [13]. In contrast, our attack detects keystrokes from an application's execution, which is much more reliable, and also works when the victim uses the system locally. Actually, we can do more with an application's semantic information recovered from its executable and procfs. For example, once we observe that the same user runs the command `su` multiple times through SSH, we can assume that the key sequences she entered in these interactions actually belong to the same password, and thus accumulate their timing sequences to infer her password, which is more effective than using only a single sequence as the prior work [26] does. As another example, we can even tell when a user is typing her username and when she inputs her password if these two events have different ESP/EIP patterns in an application.

This paper makes the following contributions:

- *Novel techniques for determining inter-keystroke timings.* We propose a suite of new techniques that accurately detects keystrokes and determines inter-keystroke timings on Linux. Our approach includes an automatic program analyzer that extracts from the binary executable of an application the instructions related to keystroke events, which are used to build a pattern that fingerprints the events. During the execution of the application, we use a shadow program to log a trace of its ESP/EIP values from procfs. The trace is searched for the occurrences of the pattern to identify inter-keystroke timing. Our attack does not need to change the application under surveillance, and works even in the presence of address space layout randomization [29] and realistic workloads. Our research also demonstrates that though other UNIX-like systems (e.g., FreeBSD and OpenSolaris) do not publish these register values, they are subject to similar attacks that utilize other information disclosed by their procfs.
- *Keystroke analysis.* We augmented the existing keystroke analysis technique [26] with semantic information: once multiple timing sequences are found to be associated with the same sequence of keys, our approach can combine them together to infer these keys, which turns out to be very effective. We also took advantage of the information regarding the victim's writing style to learn the English words she types.
- *Implementation and evaluations.* We implemented an automatic attack tool and evaluated it using real applications, including `vim`, `SSH` and `Gedit`. Our experimental study demonstrates that our attack is realistic: inter-keystroke timings can be reliably collected even when the system is under a realistic workload. We also discuss how to defend against this attack.

The attack we propose aims at keystroke eavesdropping. However, the privacy implication of disclosing the ESP/EIP information of other users' process can be much more significant. With our techniques, such information can be conveniently converted to a system-call sequence that describes the behavior of the process, and sometimes, the data it works on and the activities of its users. As a result, sensitive information within the process can be inferred under some circumstances: for example, it is possible to monitor a key-generation program to deduce the secret key it creates for another user, because the key is computed based on random activities within a system, such as mouse moves, keystrokes and networking events, which can be discovered using our techniques.

The information-leak vulnerability exploited by our attack is pervasive in Linux: we checked 8 popular distributions (Red Hat Enterprise, Debian, Ubuntu, Gentoo, Slackware, openSUSE, Mandriva and Knoppix) that represent the mainstream of Linux market [9] and found that all of them publish ESP and EIP. Some other Unix-

like systems, particularly FreeBSD, have different implementations of procfs that do not disclose the contents of those registers to unauthorized party. However, given unrestricted access to procfs, similar attacks that use other information can still happen: for example, we found that `/proc/pid/status` on FreeBSD reveals the accumulated kernel time consumed by the system calls within a process; such data, though less informative than ESP/EIP, could still be utilized to detect keystrokes in some applications, as discussed in Section 6.2. Fundamentally, we believe that the privacy risks of procfs need to be carefully evaluated on multi-core systems, as these systems enable one process to gather information from other processes in real time.

The rest of the paper is organized as follows. Section 2 presents an overview of our attack. Section 3 elaborates our techniques for detecting inter-keystroke timings. Section 4 describes a keystroke analysis using the timings. Section 5 reports our experimental study. Section 6 discusses the limitations of our attack, similar attacks on other UNIX-like systems and potential defense. Section 7 surveys the related prior research, and Section 8 concludes the paper.

2 Overview

This section describes our attack at a high level.

Attack phases. Our attack has two phases: first, the timing information between keystrokes is collected, and then such information is analyzed to infer the related key sequences. These phases and their individual components are illustrated in Figure 1. In the first phase, our approach analyzes the binary executable of an application to extract the ESP/EIP pattern that characterizes its response to a keystroke event, and samples the `stat` file of the application at its runtime to log a trace of those register values. Inter-keystroke timings are determined by matching the pattern to the trace. In the second phase, these timings are fed into an analysis mechanism that uses the Hidden Markov Model (HMM) to infer the characters being typed.

An example. We use the code fragment in Figure 2 as an example to explain the design of the techniques behind our attack. The code fragment is part of an editor program[1] for processing a keystroke input. Upon receiving a key, the program first checks its value: if it is '`MOV_CURSOR`', a set of API calls are triggered to move the cursor; otherwise, the program makes calls to insert the input letter to the text buffer being edited and display its content. These two program behaviors produce two different system call sequences, as illustrated in the figure. This example is written in `C` for illustration purpose. Our techniques actually work on binary executables.

```
1 ...
2 if (input_ready()) {
3   c = vgetc();
4   switch (c) {
5     ...
6     MOV_CURSOR: {
7       ...
8       cursor_pos_info();
9       update_cursor();
10      ...
11        ...
12      };
13      default: { //insert a char
14        ...
15        alloc_buf();
16        insert_char();
17        update_undo();
18        flush_buffers;
19        ...
20    }
21  }
22  ...
23 }
```

System Call Sequence for MOV_CURSOR:	System Call Sequence for insert a char:
read	read
select	select
select	select
select	select
select	select
select	select
select	select
select	select
select	select
select	write
write	select
select	_llseek
selec	write
	select
	fsync
	select
	select

Figure 2: An Example.

To prepare for an attack, our approach first performs a dynamic analysis on the program's executable to extract its ESP/EIP pattern that characterizes the program's response to a keystroke input. Examples of such a response includes allocating a buffer to hold the input (`alloc_buf()`) and inserting it to the text (`insert_char()`). In our research, we found that such a pattern needs to be built upon system calls because sampling of a process's `stat` file can hardly achieve the frequency necessary for catching the ESP/EIP pairs unrelated to system calls (Section 3.1). When a system call happens, the EIP of the process always points to virtual Dynamic Shared Object (vDSO)[2] [22], a call entry point set by the kernel, whereas its ESP value reflects the dynamics of the process's call stack. Therefore, our approach uses the ESP sequence of system calls as the pattern for keystroke recognition. Such a pattern is automatically identified from the executable through a differential analysis or an instruction-level program analysis (Section 3.1).

When the program is running on behalf of the victim, our approach samples its `stat` file to get its ESP/EIP values, from which we remove those unrelated to system calls according to their EIPs. The rest constitutes an ESP trace of the program's system calls. This trace is searched for the ESP patterns of keystrokes. Note that the trace may only contain part of the patterns: in the example, inserting a character triggers 17 system calls, whereas only 5 - 6 of them appear in the trace. Our approach uses a threshold to determine a match (Section 3.3). Inter-keystroke timings are measured between two successive occurrences of a same pattern.

The timings are analyzed using an n-Viterbi algorithm [26] to infer the characters being typed: our approach first constructs an HMM based upon a set of train-

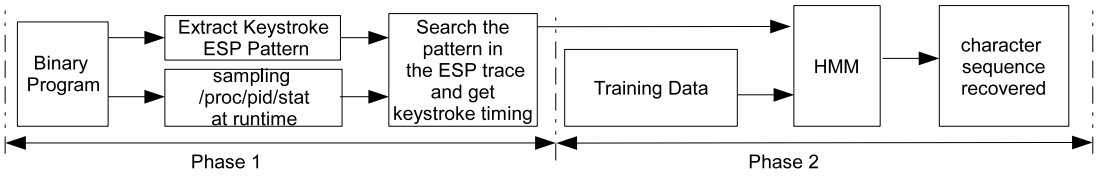

Figure 1: Attack phases

ing data that reflect the timing distributions of different key pairs the victim types, and then runs the algorithm to compute n most likely key sequences with regards to the timing sequence acquired from the ESP trace. We extend the algorithm to take advantage of multiple traces of the same key sequence, which turns out to be particularly effective for password cracking. We also show that the techniques are also effective in inferring English words a user types.

Assumptions. We made the following assumptions in our research:

- *Capability to execute programs.* To launch the attack, the attacker should own or control an account that allows her to execute her programs. This is not a strong assumption, as most users of UNIX-like systems do have such a privilege. The attacker here could be a malicious insider or an intruder who cracks a legitimate user's account.
- *Multi-core systems.* To detect a keystroke, our shadow process needs to access the ESP of the target process before it accomplishes key-related system calls. However, due to process scheduling, this is not very likely to happen on a single-core system. On one hand, these system calls are typically done within a single time slice. On the other hand, the shadow process often lacks sufficient privileges to preempt the target process when it is working on keystroke inputs, as the latter is usually granted with a high privilege during its interactions with the user. As a result, our process can become completely oblivious to the keystroke events in the target process. This problem is effectively avoided on a multi-core system, which allows us to reliably detect keystroke events in the presence of realistic workloads[3], as observed in our experiment (Section 5). Given the pervasiveness of multi-core systems nowadays, we believe that the assumption is reasonable.
- *Access to the victim's information.* Our attack requires a read access to the victim's procfs files. This assumption is realistic for Linux, on which most part of procfs are readable for every user by default. Though one can change her files' permissions, this can hardly eliminate the problem: all the procfs files are dynamically created by the kernel when a new process is forked and their default permissions are also set by the kernel; as a result, one needs to revise these permissions as soon as she triggers new process, which is unreliable and also affects the use of the tools such as `top`. The fundamental solution is to patch the kernel, which has not been done yet. In addition, we assume that the attacker can obtain some of the text the victim types as training data. This is possible on a multi-user system. For example, some commands typed by a user, such as "`su`" and "`ls`", causes new processes to be forked and therefore can be observed by other users of the system, which allows the observer to bind the timing sequence of the typing to the content of the text the user entered. As another example, a malicious insider can use the information shared with the victim, such as the emails they exchanged, to acquire the latter's text and the corresponding timings.

3 Inter-keystroke Timing Identification

In this section, we elaborate our techniques for obtaining inter-keystroke timings from a process.

3.1 Pattern Extraction

The success of our attack hinges on accurate identification of keystroke events from the victim's process. We fingerprint such an event with an ESP pattern of the system calls related to a keystroke. The focus on system calls here comes from the constraints on the information obtainable from a process: on one hand, a significant portion of the process's execution time can be spent on system calls, particularly when I/O operations are involved; on the other hand, our approach collects the process's information through system calls and therefore cannot achieve a very high sampling rate. As a result, the shadow program that logs ESP/EIP traces is much more likely to pick up system calls than other instructions. In our research, we found that more than 90% of the ESP/EIP values collected from a process actually belong to system calls. Note that a process's EIP when it is making a system call always points to vDSO. It is used in our research to locate the corresponding ESP whose content is much more dynamic and thus more useful for fingerprinting a keystroke event.

Our approach extracts the ESP pattern through an automatic analysis of binary executables. This analysis is conducted offline and in an environment over which the attacker has full control. Following we present two analysis techniques, one for the programs that execute in a deterministic manner and the other for those whose executions are affected by some random factors.

Differential analysis. Many text-based applications such as `vim` are deterministic in the sense that two independent runs of these applications under the same keystroke inputs yield identical system call traces and ESP sequences. The ESP patterns of these applications can be easily identified through a differential analysis that compares the system call traces involving keystroke events with those not. Specifically, our program analyzer uses `strace` [27] to intercept the system calls of an application and record their ESP values when it is running. An ESP sequence is recorded before a keystroke is typed, and another sequence is generated after the keystroke occurs[4]. The ESP pattern for a keystroke event is extracted from the second sequence after removing all the system calls that happen prior to the keystroke, as indicated by the first sequence. To ensure that the pattern does not contain any randomness, we can compare the ESP trace of typing the same character twice with the one involving only a single keystroke to check whether the ESPs associated with the second keystroke are identical to those of the first one. The same technique is also applied to test different keys that may have discrepant patterns. In the example described in Figure 2, the ESP sequence of `vim` before Line 2 is dropped from the traces involving keystrokes and as a result, the system calls triggered by the instructions from Line 7 to 11 are picked out as the fingerprint for '`MOV_CURSOR`' and those between Line 14 and 19 identified as the pattern for inserting a letter.

The ESP pattern identified above will go through a false positive check to evaluate its accuracy for keystroke detection. In other words, we want to know whether the pattern or a significant portion of it can also be observed when the user is not typing. This is achieved in our research through searching for the pattern in an application's ESP trace unrelated to keystroke inputs. Specifically, our analyzer logs the execution time between the first and the last system calls on the pattern, and uses this time interval to define a duration window on the trace, which we call *trace window*. The trace window is slid on the trace to determine a segment against which the pattern is compared. For this purpose, every ESP value on the trace is labeled with the time when its corresponding system call is invoked. The trace window is first located prior to the first ESP value on the trace. Then, it is slid rightwards: each slide either moves an ESP into the window or moves one outside the window. After a slide, our analyzer attempts to find the longest common sequence between the trace segment within the window and the pattern. This is the well-known LCS problem [4], which can be efficiently solved through dynamic programming [15]. The size of such a sequence, which we call an *FP level*, is recorded. As such, our approach keeps on sliding the trace window to measure FP levels until all the ESP values on the trace have left the window.

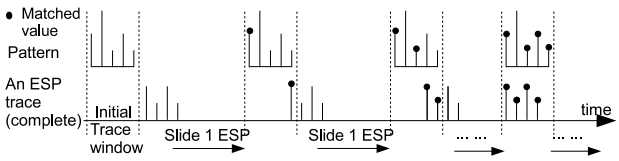

Figure 3: A false positive check. Spikes in the figure represent ESP values.

Figure 3 presents an example that shows how the algorithm works. In the initial state, the trace window is located before the first ESP value. Then the trace window starts to slide right to include the first ESP value, which gives a FP level of one. After the window slides again to include one more ESP value, our algorithm returns a common sequence with two members. This process continues, and finally, the window is moved to embrace all four trace members and we observe an FP level of four. This algorithm identifies the portion of the pattern that can show up in absence of keystrokes. The size of the portion, as indicated by the FP level, is used to determine a threshold for recognizing keystrokes from an incomplete ESP trace sampled from a process, which is elaborated in Section 3.3.

Instruction-level analysis. Applications with graphic user interfaces (GUI) can work in a non-deterministic manner: these applications are event-driven and can change their system-call behaviors in response to the events from operating systems (OS), which can be unpredictable. For example, `Gedit` uses a timer to determine when to flash its cursor; the timer, however, can be delayed when the process is switched out of the CPU, which causes system call sequences to vary in different runs of the application. To extract a pattern from these applications, we adopted an instruction-level analysis as described below.

Under Linux, many X-Window based applications are developed using the GIMP Toolkit (aka. GTK+) [28]. GTK+ uses a standard procedure to handle the keystroke event: a program uses a function such as `gtk_main_do_event(event)` to process `event`; when a key is pressed[5], this function is invoked to trigger a call-back function of the keystroke event. In our research, we implemented a Pin [20] based analysis tool that automatically analyzes a binary executable at the instruction level to identify such a function. After a key has been typed, our analyzer detects the keystroke

event from the function's parameter and from that point on, records all the system calls and their ESPs until the executable is found to receive or dispatch a new event, as indicated by the calls to the functions like `g_main_context_acquire()`. All these system calls are thought to be part of the call-back function and therefore related to the keystroke event[6]. The pattern for keystroke recognition is built upon these calls. We also check false positives of the pattern, as described before.

3.2 Trace Logging

Our attack eavesdrops on the victim's keystrokes through shadowing the process that receives her keystroke inputs. Our shadow process stealthily monitors the target process's keystroke events by keeping track of its ESP/EIP values disclosed by its `stat` file. Since the attack happens in the userland, the attacker has to use system calls to open and read the file. Moreover, a more efficient approach, memory mapping through `mmap()`, does not work on the virtual file that exists only in memory. These issues prevent the shadow process from achieving a high sampling rate. For example, a program we implemented for evaluating our approach updated ESP/EIP values every 5 to 10 microseconds. As a result, we could end up with an incomplete ESP/EIP trace of the target process. This, however, is sufficient for determining inter-keystroke timings, as we found in our research (Section 3.3).

Trace logging with full steam can cost a lot of CPU time. If the activity drags on, suspicions can be roused and alarms can be triggered. To avoid being detected, our attack takes advantage of the semantic information recovered from procfs and the target application to concentrate the efforts of data collection on the time interval when the victim is typing the information of interest to the attacker. For example, the shadow process starts monitoring the victim's `SSH` process at a low rate, say once per 100 milliseconds; once the process is observed to fork a `su` process, our shadow process immediately increases its sampling rate to acquire the timings for the password key sequence. Another approach is using an existing technique [32] to hide CPU usage: UNIX-like systems keep track of a process's use of CPU according to the number of ticks it consumes at the end of each tick; the trick proposed in [32] lets the attack process sleep just before the end of each tick it uses and as a result, OS will schedule a victim process to run and bill the whole tick to that victim process instead of the attack process. We implemented this technique and found that it was very effective (Section 5).

3.3 Timing Detection

We determine inter-keystroke timings from the time intervals between the occurrences of a pattern on an ESP trace sampled from an application's system calls. Two issues here, however, complicate the task. First, some Linux versions may run the mechanisms for address space layout randomization (ASLR) [29] that can cause the ESP values on the pattern to differ from those on the trace. Second, the trace can be incomplete, containing only part of the system calls on the pattern, which makes recognition of the pattern nontrivial. Following we show how these issues were handled in our research.

ASLR performed by the tools such as `Pax` [30] involves randomly arranging the locations of an executable's memory objects such as stack, executable image, library images and heap. It is aimed at thwarting the attacks like control-flow hijacking that heavily rely on an accurate prediction of target memory addresses. Though the defense works on the attacks launched remotely, it is much less effective on our attack, which is commenced locally. Specifically, the address for the bottom of a process's stack can be found in its `stat` and `/proc/PID/maps`[7]. This allows us to "normalize" the ESP values on both the trace and the pattern with the differences between the tops of the stack, as pointed by the ESPs, and their individual bottoms. Neither does ASLR prevent us from correlating an ESP/EIP pair on a trace to a system call, though the knowledge about the vDSO address may not be publically available on some Linux versions: we can filter out the pairs unrelated to system calls according to the observation that the vast majority of the members on the trace actually belong to system calls and therefore have the same EIP values.

To recognize an ESP pattern from an incomplete ESP trace of system calls, we use a threshold τ: a segment of the trace, as determined by the trace window, is deemed matching the pattern if it contains at least τ ESP values of system calls and the sequence of these values also appear on the pattern. The threshold here can be determined using the results of the false positive test described in Section 3.1. Let h be the highest FP level found in the test, and s be the number of the system calls that our shadow process can find from a process when a keystroke occurs. We let $\tau = h + 1$ if $s > h$. Intuitively, this means that a trace segment is considered matching the pattern if it does not contain any ESP sequences not on the pattern and no segments unrelated to keystrokes can match as many ESP values on the pattern as that segment does[8]. If $s \leq h$, we have to set $\tau = s$ because we cannot get more than s ESP samples for every keystroke when monitoring a process. Several measures can be taken to mitigate the false positives that threshold could bring in. One approach is to leverage the observation that people typ-

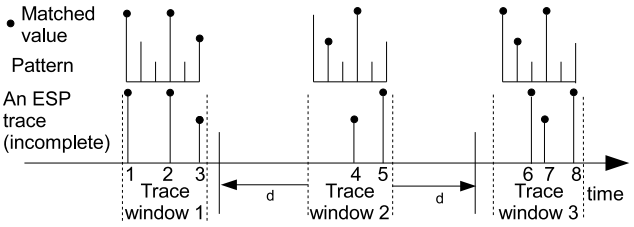

Figure 4: Using time frame d to remove possible false positive matches

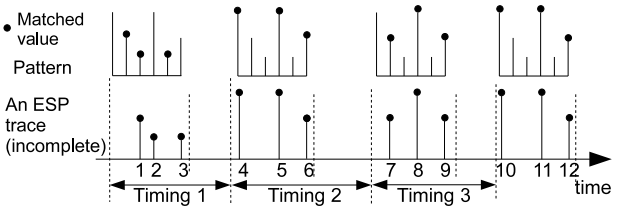

Figure 5: Pattern matching on an ESP trace and the timing interval

ically type more than one key within a short period of time. Therefore, we can require that a segment matching a pattern according to τ be preceded or followed by another pattern-matching segment within a predetermined time frame d, before both of them can be deemed to be indicative of keystroke events. Figure 4 presents an example in which the segment within the Window 2 is not treated as a match to the pattern because there is no other matches happening within the time frame d either before or after the window. In another approach, we use the execution time of a process to estimate the time point when it starts receiving keystrokes, which helps avoid searching the trace unlinked to keystrokes.

After normalizing ESP values and determining the threshold τ, our approach starts searching the trace sampled from the victim's process for the occurrences of the pattern. The searching algorithm we adopted slides the trace window in the same way as the false positive check does (Section 3.1). For each slide, an LCS problem is solved to find the longest common sequence between the trace segment in the window and the pattern. If the length of the sequence is no less than τ and every member on the segment is also on the sequence, the segment is labeled as a match. Once a match is found, we slide the window rightwards to pass all trace members within a short time interval that describes the minimal delay between two consecutive keystrokes, and then start the next round of searching. This process continues until all trace members pass the window. Then, our approach determines timings from the segments labeled as matches: the time interval between two such segments is identified as an inter-keystroke timing if there is no other labeled segments in-between and the duration of the interval is below a predetermined threshold that serves to rule out the long latencies caused by intermittent typing. An example for illustrating the algorithm is presented in Figure 5, in which the trace window locates four matches with $\tau = 3$, and the durations between these matches are picked out as inter-keystroke timings.

4 Keystroke Analysis

In this section, we describe how to use inter-keystroke timings to infer the victim's key sequence. Our approach is built upon the technique used in the existing timing attack [26]. However, we demonstrate that the technique can become much more effective with the information available on a multi-user system.

4.1 HMM-based Inference of Key Sequences

A Hidden Markov Model [24] describes a finite stochastic process whose individual states cannot be directly observed. Instead, the outputs of these states are visible and therefore can be used to infer the existence of these states. An HMM, like a regular Markov model, assumes that the next states a system can move into only depend on the current state. In addition, it has a property that the outputs of a state are completely determined by that state. These two properties allow a hidden sequence to be easily computed and therefore make the model a pervasive tool for the purposes such as speech recognition and text modeling.

Prior research [26] models the problem of key inference using an HMM. Specifically, let $K_0, \ldots, K_T$ be the key sequence typed by the victim, and $q_t \in Q$ ($1 \leq t \leq T$) be a sequence of states representing the key pair (K_{t-1}, K_t), where Q is the set of all possible states. In each state q_t, an inter-keystroke latency y_t with a Gaussian-like distribution can be observed. Our objective is to find out the hidden states $(q_1, \ldots, q_T)$ from the timings $(y_1, \ldots, y_T)$. This modeling is simple and was shown to work well in practice [26], and is further confirmed by our research, though it has oversimplified the relations between the characters being typed: particularly, the chance for a letter to appear at a certain position in an English word may actually relate to all other letters before it, which invalidates the HMM assumption that a transition from q_t to q_{t+1} depends only on q_t.

The HMM for key inference can be solved using the Viterbi algorithm [24], a dynamic programming algorithm that computes the most likely state sequence $(q_1, \ldots, q_T)$ from the observed timing sequence $(y_1, \ldots, y_T)$. Let $V(q_t)$ be the probability of the sequence that most likely ends in q_t at time t. The algorithm computes $V(q_t)$ through two steps. In the first step, we assign a set of initial probabilities $V(q_1) =$

$Pr[q_1|y_1]$. The second step inductively computes $V(q_t)$ for every $1 < t \leq T$ and every $q_t \in Q$ as $V(q_t) = \max_{q_{t-1}} Pr[y_t|q_t]Pr[q_t|q_{t-1}]V(q_{t-1})$, where $Pr[y_t|q_t]$ can be estimated from a set of training data (the third assumption in Section 2) and $Pr[q_t|q_{t-1}]$, the transition probability, comes from a uniform distribution over the states reachable from q_{t-1}. This step also keeps track of all the prior states on the sequence with the probability $V(q_t)$. The most likely sequence is identified from the state q_T that maximizes $V(q_T)$. A direct application of this approach, however, does not work well in practice, because even the most likely sequence usually has a very small probability to match the real keystroke inputs. This problem is mitigated in the prior work [26] that extends the algorithm to the n-Viterbi algorithm so as to return the top n most likely sequences given a timing sequence. The difference here is that the n-Viterbi algorithm changes the inductive step (the second step) to identify the sequences with the n largest probabilities. The details of the algorithm can be found in [26].

4.2 Password Cracking

The effectiveness of the n-Viterbi algorithm can be significantly improved with the information available on a multi-user system. Particularly, the name of a process and its owner can be directly found from procfs or indirectly from running commands such as `ps` or `top`. Once the same user is observed to run the same application multiple times and if such interactions happen within a no-so-long period of time and all involve typing passwords, a reasonable assumption we can make is that all these passwords are actually the same. Therefore, we can combine together the timing sequences recorded from individual interactions to infer a key sequence. Following we describe two ways to do that.

Our first approach is simply averaging all the timings for every key pair to create a new sequence and run the n-Viterbi algorithm over it. Formally, given m timing sequences $(y_1^1, \ldots, y_T^1), \ldots, (y_1^m, \ldots, y_T^m)$, we can compute a new sequence $(y_1, \ldots, y_T)$, where $y_t = \frac{1}{m}\sum_{1\leq i\leq m} y_t^i$ and $1 \leq t \leq T$. The rationale here is that the distribution of the timing y_t^i of a key pair q_t is a Gaussian-like unimodal distribution and therefore the probability $Pr[y_t|q_t]$ in the inductive step of the algorithm is maximized when y_t becomes the mean of the distribution, which is approximated by averaging all y_t^i. This approach works particularly well when the means of two key pairs are not extremely close.

The other approach, which we call the m-n-Viterbi algorithm, utilizes multiple observations to perform the inductive step of the original algorithm. Specifically, our approach replaces $Pr[y_t|q_t]$ in that step with $Pr[y_t^1, \ldots, y_t^m|q_t] = Pr[y_t^1|q_t] \ldots Pr[y_t^m|q_t]$ given these observations $(y_t^1, \ldots, y_t^m)$ are independent from each other. This treatment works even in the presence of the key pairs with very close timing distributions. However, it needs a large number of timing sequences to get a good outcome.

Our research shows that both approaches can significantly shrink the space for searching a password. Actually, in our experiment (Section 5.2), we found that using 50 timing sequences, our techniques sped up the password searching by factors ranging from 250 to 2000.

4.3 English Text

Recovery of English text from a timing sequence is no less challenging than password cracking. A password can be figured out through testing many candidates against the target application or a hashed password list. However, the same trick cannot be played on English words because no application and password list can tell you whether you made a right guess. All that we can do is to check all the combinations of the possible words to see whether a meaningful sentence comes out, which becomes a daunting task if the list of such words is long. Moreover, it can be more difficult to find multiple timing sequences associated with the same text, and therefore the aforementioned approaches become less applicable. On the bright side, English words are much less random than passwords: the letters they include and the combinations of those letters have distributions with low entropies. Such a property can be leveraged to adjust the transition probabilities of an HMM to improve the outcomes of key sequence inference. Here we elaborate such techniques used in our research.

A prominent property of English text is use of the SPACE character to separate words. People tend to type the letters in a word faster than SPACE, a signal for a transition between words. This gives the character an identifiable timing feature: typically the key pair involving SPACE incurs longer inter-keystroke latency than other pairs, as illustrated in Figure 6. In our research, we detected SPACE by checking if the timing interval is larger than a predetermined threshold. This threshold can be determined from the training data collected from the victim's typing. Knowledge about the SPACE key helps us to divide a long timing sequence into a collection of small sequences, with each of them representing a word, and then learn these words one by one.

Another important property of English text is its distinct distribution of letters. It is well known that some letters such as 'e' occur more frequently than others, and some bigrams like 'th' and trigrams like 'ion' are also pervasive in a meaningful text. This fact has been leveraged by frequency analysis to crack classic ciphers [1]. The same game can also be played to make key se-

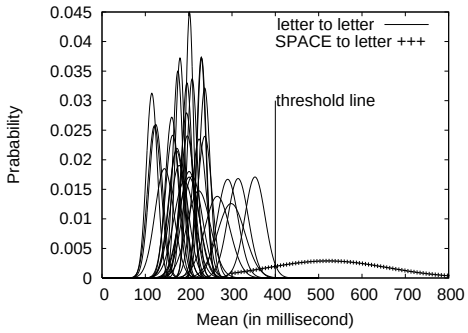

Figure 6: Timing Distribution of SPACE-letter pair, letter-letter pair and threshold

quence inference more effective: we can adjust the transition probabilities of an HMM to ensure that the transition between certain states such as ('i','o') to ('o', 'n') is more likely to happen than others. These probabilities can be conveniently obtained from various public sources [18, 10] that provide the statistics of common English text. Such statistics can be further tuned to the victim's writing style according to public writing samples such as her web pages and publications. Moreover, it comes with no surprise that users on the same system are often related: for example, they could all belong to one organization. This allows the attacker to get familiar with the victim's writing from the information they exchanged, for example, the emails between them. In addition, since the timing sequence corresponding to such information can also be identified using our technique, the attacker can actually use the information as the training data for estimating the timing distributions of different key pairs the victim typed.

5 Evaluation

In this section, we describe an experimental study of the attack techniques we propose. Our objective is to understand whether these techniques present a realistic threat. To this end, we evaluated them using 3 common Linux applications: `vim`, `SSH` and `Gedit`. In our experiments, we first ran our approach to automatically extract timing sequences when a user was typing, evaluated the accuracy of these timings and the effectiveness of the attack under different workloads. Then, we analyzed them using our techniques to study how much keystroke information could be deduced. Our experiments were mainly carried out on a computer with a 2.40GHz Core 2 Duo processor and 3GB memory, on which we conducted our study under three Linux versions: RedHat Enterprise Linux 4.0, Debian 4.0 and Ubuntu 8.04. We found that our techniques worked effectively even in the presence of realistic workloads on the server. This suggests that the information leaks caused by procfs can be a real security problem.

Table 1: Normalized ESP pattern values (include system calls)

vim		ssh		gedit	
SysCall	ESP	SysCall	ESP	SysCall	ESP
read	1628	rt_sigprocmask	4932	gettimeofday	3624
select	1604	rt_sigprocmask	4932		
select	1876	read	20908		
select	2244	select	4548		
select	1540	rt_sigprocmask	4932		
select	1908	rt_sigprocmask	4932		
select	1556	write	37436		
select	1924	ioctl	37500		
select	1604	select	4548		
write	1548	rt_sigprocmask	4932		
select	1972	rt_sigprocmask	4932		
_llseek	1876	read	37436		
write	1836	select	4548		
select	2180	rt_sigprocmask	4932		
fsync	1752	rt_sigprocmask	4932		
select	2148	write	4620		
select	1972	select	4548		

5.1 Inter-keystroke Timings

As the first step of our evaluation, we applied our technique to identify the timings from `vim`, `SSH` and `Gedit` on a multi-core system.

`vim`. `vim` is an extremely common text editor, which is supported by almost all Linux versions. It fits well with the notion of deterministic programs as discussed in Section 3.1, because independent runs of the application with the same inputs always produce the same system call sequence and related ESP sequence. This property enabled us to identify its ESP pattern for a keystroke event using the differential analysis. The pattern we discovered for inserting a letter includes 17 calls. These calls and their normalized ESP values are presented in Table 1. We further ran the application from a user account to enter words, and in the meantime, launched a shadow process from another account to collect the ESP trace of the application. From the trace, our approach automatically identified all the keystrokes we typed. Table 2 shows a trace segment corresponding to two keystrokes, which involves 5 system calls for each keystroke.

In order to evaluate the accuracy of the timing sequence our shadow process found, an instrumented version of `vim` was used in our experiment, which recorded the time when it received a key from `vgetc()`. Such information was used to compute a real timing sequence. We compared these two sequences and found that the de-

Table 2: Examples of ESP traces (values that appear in the pattern are in bold font).

vim	ssh	gedit
1604	**4548**	520
2244	**4932**	2988
1908	**20908**	3052
1924	**4548**	696
1972	**37500**	**3624**
1604	**4548**	3068
2244	**37436**	2988
1908	**4932**	696
1924	**4620**	520
1972	**4548**	2988

viations between corresponding timings were at most 1 millisecond, below 3% of the average standard deviation of the timings of different key pairs, as illustrated in Table 3. This demonstrates that the timings extracted from the process were accurate.

SSH. The Secure Shell (SSH) has long been known to have a weakness in its interactive mode, where every keystroke is transmitted through a separate packet and immediately after the key is pressed. This weakness can be exploited to determine inter-keystroke timings for inferring the sensitive information a user types, such as the password for `su`. Prior work [26] proposes an attack that eavesdrops on an SSH channel to identify such timings. A problem of the attack, as pointed out by SSH Communications Security, is that determination of where a password starts in an encrypted connection can be hard [25]. This problem, however, does not present a hurdle to our attack, because we can easily find out from procfs when `su` is spawned from an SSH process, and start collecting information from SSH from then on. This is exactly what we did in our experiment.

Using the differential analysis, our approach automatically discovered an ESP pattern from SSH when a key was typed for entering a password for `su`. We further ran a shadow process to monitor another user's SSH process: as soon as it forked an `su` process, our shadow process started collecting ESP values from the SSH process's `stat` file. The trace collected thereby was compared with the pattern to pinpoint keystroke events and gather the timings between them. The pattern that we found in our experiment included 17 system calls, of which 7 to 10 appeared in every occurrence of the pattern on the trace. The detailed experimental results are in Table 1 and Table 2.

Verification of the correctness of those timings turned out to be more difficult than we expected. `su` does not read password characters one by one from the input. Instead, it takes all of them after a RETURN key has been stroked. Therefore, instrumentation of its source code will not give us the real timing sequence. We solved this problem by replacing `su` with another program that recorded the time when it received a key from SSH, and used such information to generate a timing sequence. This sequence was found to be very close to the one we got from the trace collected by our shadow process, as described in Table 3. We further employed the timings obtained from `su` to infer the passwords being typed, which we found to be very effective (Section 5.2).

Table 3: Examples of the timings measured from ESP traces (Measured) and the real timings (Real) in milliseconds.

Timings	`vim`		`ssh`		`Gedit`	
	measured	real	measured	real	measured	real
1	80	81	135	135	301	303
2	139	139	124	123	285	285
3	88	88	103	103	259	259
4	101	101	110	109	236	236
5	334	335	134	134	181	182
6	86	87	111	110	265	265
7	124	124	132	132	174	174

`Gedit`. `Gedit` is a text editor designed for the X Window system. Like many other applications based upon the GTK+, it is non-deterministic in the sense that two independent runs of the application under the same inputs often produce different system call sequences. In our experiment, we performed an instruction-level analysis of its binary executables using the Pin-based tool we developed. This analysis revealed the call-back function of the key-press event, from which we extracted the system call sequence and related ESP sequence. An interesting observation is that `Gedit` actually does not immediately display a character a user types: instead, it put the character to a buffer through a GTK+ function `gtk_text_buffer_insert_interactive_at_cursor()`, which does not involve any system calls, and the content of the buffer is displayed when it becomes full or a timer expires. As a result, we could not count on the system calls involved in such a display process for fingerprinting keystrokes. Actually, only one system call was found to be present every time when a key was received: `gettimeofday()`, a call that `Gedit` uses to determine when to auto-save the document the user is editing. This call seems too general. However, its ESP value turned out to be specific enough for a pattern: in our false positive check, we did not find any other system calls within the application that also had the same ESP. Moreover, our shadow process always caught that ESP whenever we typed. Therefore, this ESP value was adopted as the pattern in our experiment. We further instrumented `Gedit` to dump the time when this call was invoked for calculating the real timing sequence. Table 1 shows that this sequence is very close to the one collected by our shadow process.

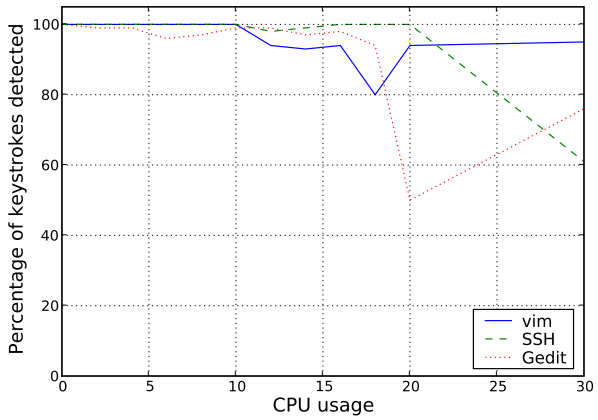

Figure 7: Percentage of keystrokes detected vs. CPU usage

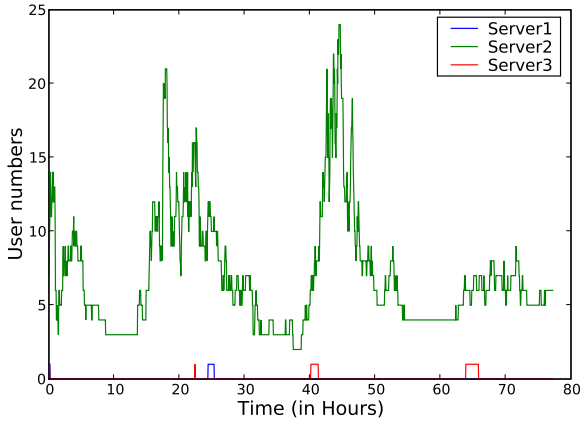

Figure 9: Variations of user numbers on the three servers during 72 hours

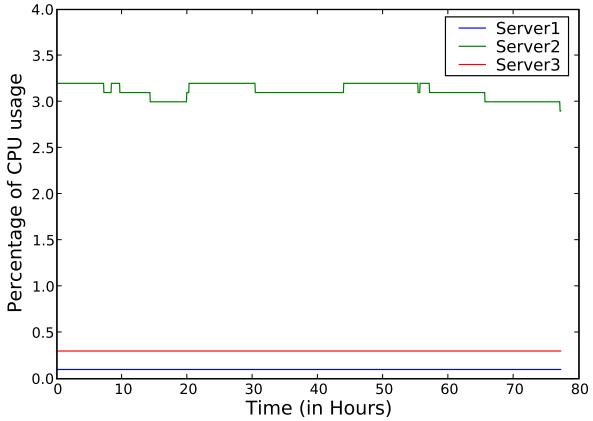

Figure 8: CPU usages of three real-world servers during 72 hours

Impacts of server workloads. A multi-user system often concurrently serves many users. These users' activities could interfere with the collection of inter-keystroke timings. This problem was studied in our research through evaluating the effectiveness of our attack under different workloads. Specifically, we ran our attacks on `vim`, `SSH` and `Gedit` under different CPU usages to measure the percentage of the keystrokes still detectable to our shadow process. The experimental results are elaborated in Figure 7. Here, we sketch our findings.

We found that the impacts of workloads varied among applications. The attacks on `vim` and `SSH` appear to be quite resilient to the interferences from other processes: our shadow process picked up 100% keystrokes for both applications when CPU usage was no more than 10% and still detected 94% from `vim` when the usage went above 20%. In contrast, the attack on `Gedit` was less robust: we started missing keystrokes when more than 2% of CPU time was consumed by other processes. This discrepancy comes from applications' ESP patterns: those involving more system calls are easier to detect.

On the other hand, the workloads on a real-world system are reasonable enough to be handled by our attack. Figure 8 and 9 reports the CPU usages and user numbers we measured from three real-world systems, including a Linux workstation in a public machine room (Server 1), a server for students' course projects (Server 3) and a web server of Indiana University that allows SSH connections from its users (Server 2). The number of users on these systems range from 1 to 24. Our 72-hour monitoring reveals that for 90 percent of time, the CPU usages of these servers were below 3.2%.

We also implemented the technique proposed in [32] to hide the CPU usage of our shadow process. As a result, the process appeared to consume 0% of CPU, as observed from `top`. The cost, however, was that it only reliably identified about 50% of keystrokes we entered. Nevertheless, this still helped inference of keys, particularly when the same input from a user (e.g., password) was sampled repeatedly, as discussed in Section 4.2.

5.2 Key Sequence Inference

We further studied how to use the timings to infer key sequences. Experiments were conducted in our research to evaluate our techniques using both passwords and English words. Here we report the results.

Password. To study the effectiveness of our approach on passwords, we first implemented the n-Viterbi algorithm [26] and used it to compute a baseline result, and then compared the baseline with what can be achieved by the analysis using multiple timing sequences, as described in Section 4.2. Our experiment was carefully

Table 4: The percentage of the search space the attacker has to search before the right password is found.

Method	Test Cases		
	password 1	password 2	password 3
Baseline(n-Viterbi)	7.8%	6.6%	6.8%
Timing Averaging	0.38%	0.34%	0.05%
m-n-Viterbi	0.39%	0.34%	0.05%

designed to make it comparable with that of the prior work [26]: we chose 15 keys for training and testing an HMM, which include 13 letters and 2 numbers[9]. From these keys, we identified 225 key pairs and measured 45 inter-keystroke timings for each of these pair from a user. We found that the timing for each pair indeed had Gaussian-like distributions. These distributions were used to parameterize two HMMs: one for the first 4 bytes of an 8-byte password and the other for the second half.

We randomly selected 3 passwords from the space of all possible 8-byte sequences formed by the 15 characters. For each password, we ran the n-Viterbi algorithm on 50 timing sequences. Each of these sequences caused the algorithm to produce a ranking list of candidate passwords. The position of the real password on the list describes the search space an attacker has to explore: for example, we only need to check 1012 candidates if the password is the 1012th member on the list, which reduces the search space for a 4-byte password by 50 times. To avoid the intensive computation, our implementation only output the top 4500 members from an HMM. We found that for about 75% of the sequences tested in our experiment, their corresponding passwords were among these members. In Table 4, we present the averaged percentage of the search space for finding a password.

We tested the timing averaging approach and m-n-Viterbi algorithm described in Section 4.2 with 50 timing sequences for each password, and present the results in Table 4. As the table shows, both approaches achieved significant improvements over the n-Viterbi algorithm: they shrank the search space by factors ranging from 250 to 2000. In contrast, the speed-up factor introduced by the n-Viterbi algorithm was much smaller[10].

We also found that the speed-up factors achieved by our approach, like the prior work [26], depended on the letter pairs the victim chose for her password: if the timing distribution of one pair (Figure 6) is not very close to those of other pairs, it can be more reliably determined, which contributes to a more significant reduction of searching spaces. For example, in Figure 6, a password built on the pairs whose means are around 300 milliseconds is much easier to be inferred than the one composed of the pairs around 100 milliseconds, as the latter pairs are more difficult to distinguish from others with very similar distributions. It is important to note that those distributions actually reflect an individual's typing practice, and therefore, the same password entered by one can become easier to crack than by another.

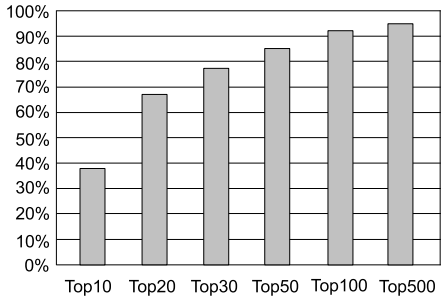

Figure 10: The success rates of the attack on English words

English words. We also studied how the timing information can help infer English words. To prepare for the experiment, a program was used to randomly generate character sequences with lengths of 3, 4 and 5 letters[11], and from them, we selected 2103 words that also appeared in a dictionary. These words were classified into three categories according to their lengths. For the words within each category, we computed a distribution using their frequencies reported by [18] . These distributions were used to determine the transition probabilities of the HMMs for individual categories, which we applied to infer the words with different lengths.

In the experiment, we randomly draw words from each category in accordance with their distribution, and typed them to collect timing sequences. The timing segments that represented individual words were identified from the sequences using the feature of the SPACE key. For each segment, we picked up an HMM according to the length of the word and solved it using the n-Viterbi algorithm, which gave us a ranking list of candidates. From the list, our approach further removed the candidates that did not pass a spelling check. We tested 14 3-letter words, 11 4-letter words and 14 5-letter words. The outcomes are described in Figure 10. From the table, we can see that the real words were highly ranked in most cases: almost 40% of them appeared in top 10 and 86% among top 50.

6 Discussion

6.1 Further Study of the Attack

Our current implementation only tracks the call-back function for the key press event. We believe that the pattern for keystroke recognition can be more specific and easier to detect by adding the ESP sequences of the system calls related to the key release event. Moreover, we evaluated our approach using three applications. It is interesting to know whether other common applications

are also subject to our attack. What we learnt from our study is that our attack no longer works when system calls are not immediately triggered by keystrokes. This could happen when the victim's process postpones the necessary actions such as access to the standard I/O until multiple keystrokes are received. For example, `su` does not read a password character by character, and instead, imports the string as a whole; as a result, it cannot be attacked when it is not used under the interactive mode of `SSH`. As another example, `GTK+` applications tend to display keys only when the buffer holding them becomes full or a timer is triggered. Further study to identify the type of applications vulnerable to our attack is left as our future research. In addition, it is conceivable that the same techniques can be applied beyond identification of inter-keystroke timing. For example, we can track the ESP dynamics caused by other events such as moving mouse to peek into a user's activities.

Our current research focuses more on extracting inter-keystroke timings from an application than on analyzing these timings. Certainly more can be done to improve our timing analysis techniques. Specifically, password cracking can be greatly facilitated with the knowledge about the types of individual password characters such as letter or number. Acquisition of such knowledge can be achieved using our enhanced versions of the n-Viterbi algorithm that accept multiple timing sequences. This "classification" attack can be more effective than the timing attack proposed in [26], as it does not need to deal with a large key-pair space. Moreover, the approach we used to infer English words is still preliminary. We did not evaluate it using long words, because solving the HMMs for these words can be time consuming. A straightforward solution is to split a long word into small segments and model each of them with an HMM, as we did for password cracking. This treatment, however, could miss the inherent relations between the segments of a word, which is important because letters in a word are often correlated. Fundamentally, the first-order HMM we adopted is limited in its capability of modeling such relations: it cannot describe the dependency relation beyond that between two key pairs. Application of other language models such as the high-order HMM [12] can certainly improve our techniques.

Actually, ESP/EIP is by no means the only information within procfs that can be used for acquiring inter-keystroke timings. Other information that can lead to a similar attack includes interrupt statistics file `/proc/interrupts`, and network status data `/proc/net`. The latter enables an attacker to track the activities of the TCP connections related to the inputs from a remote client. Moreover, the procfs of most UNIX-like systems expose the *system time* of a process, i.e., the amount of time the kernel spends serving the system calls from the process. Disclosure of such information actually enables keystroke eavesdropping, which is elaborated in Section 6.2.

6.2 Information Leaks in the Procfs of Other UNIX-like Systems

Besides Linux, most other UNIX-like systems also implement procfs. These implementations vary from case to case, and as a result, their susceptibilities to side-channel attacks also differ. Here we discuss such privacy risks on two systems, FreeBSD and OpenSolaris.

FreeBSD manages its process files more cautiously than Linux[12]: it puts all register values into the file `/proc/pid/regs` that can only be read by the owner of a process, which blocks the information used by our attack. However, we found that other information released by the procfs can lead to similar attacks. A prominent example is the system time reported by `/proc/pid/status`, a file open to every user. Figure 11 shows the correlations between the time consumed by `vim` and the keystrokes it received, as observed in our research. This demonstrates that keystroke events within the process can be identified from the change of its system time, which makes keystroke eavesdropping possible. A problem here is that we may not be able to detect special keys a user enters, for example, "`MOV_CURSOR`", which is determined from ESP/EIP information on Linux. A possible solution is using the discrepancies of system-time increments triggered by different keys being entered to fingerprint these individual keys. Further study of this technique is left to our future research.

OpenSolaris kernel makes the `/proc` directory of a process only readable to its owner, which prevents other users from entering that directory. Interestingly, some files under the directory are actually permitted to be read by others, for supporting the applications such as `ps` and `top`. Like FreeBSD, the registers of the process are kept off-limits. However, other information, including system time, is still open for grabs. Figure 11 illustrates the changes of the system time versus a series of keystrokes we entered on OpenSolaris, which demonstrates that identification of inter-keystroke timings is completely feasible on the system.

6.3 Defense

An immediate defense against our attack is to prevent one from reading the `stat` file of another user's process once it is forked, which can be done by manually changing the permissions of the file. However, this approach is not reliable because human are error-prone and whenever the step for altering permissions is inadvertently missed,

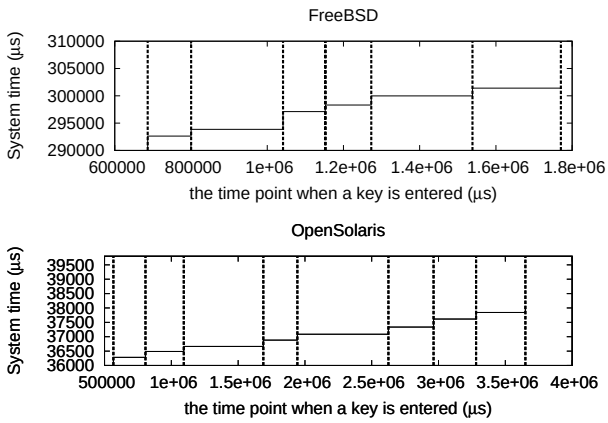

Figure 11: System time (solid line) vs. keystroke events (dashed line) in `vim` under FreeBSD (Release 7.1) and OpenSolaris (Release 2008.11). In the experiments, we found that the system time of `vim` changed only in response to keystrokes, which were recorded by shadow programs.

the door to our attack becomes wide open. The approach also affects the normal operations of common tools such as `ps` and `top`, which all depend on `stat` to acquire process information. A complete solution is to patch Linux kernels to remove the ESP and EIP information from a process's virtual file or move them into a separate file which can only be read by the owner. The problem is that there is no guarantee that other information disclosed by procfs will not lead to a similar attack (Section 6.1 and Section 6.2). Detection of our attack can also be hard, because our shadow process behaves exactly like the legitimate tools such as `top`, which also continuously read from virtual files. The shadow program can also hide its CPU usage by leveraging existing techniques [32]. Fundamentally, with the pervasiveness of multi-core systems that enable one process to effectively monitor another process's execution, we feel it is necessary to rethink the security implications of the public information available on current multi-user systems.

7 Related Work

It has long been known that individual users can be characterized by their unique and stable keystroke dynamics, the timing information that can be observed when one is typing [16]. Such information has been intensively studied for biometric authentication [21]. In comparison, little has been done to explore its potential for inferring the characters a user typed [6]. The first paper on this subject[13] proposes to measure inter-keystroke timings from the latencies between `SSH` packets [7] and use them to crack passwords. Our attack takes a different path to acquire timings: we take advantage of the information of a process exposed by procfs to find out when a key is received by the process, which has been made possible by the rapid development of multi-core techniques. Compared with the prior approach, our attack can happen to the clients who use a multi-user system locally as well as those who connect to the system remotely. Moreover, our timing analysis is much more accurate than the prior approach, through effective use of the information available from procfs. On the downside, we need a user account to launch our attack, which is not required by the prior approach. Another prior proposal measures CPU timings to acquire the information about the password a user enters [31]. This approach only gets the information such as password length and some special characters, and is subject to the interference of the activities such as processing mouse events, whereas our approach can accurately identify the events related to keystrokes and infer the characters being entered. Timing analysis has also been applied to attack cryptosystems [5, 34, 17, 8].

Keyboard acoustic emanations [34] also leak out information regarding a user's keystrokes. Such information has been leveraged by several prior approaches [2, 33, 3] to identify the keys being entered. Similar to our attack, some of these approaches also apply language models (including the high-order HMM) to infer English words. They all report very high success rates. Acoustic emanations are associated to individual keys, whereas timings are measured between a pair of keys. This makes character inference based on timings more challenging. On the other hand, acquisition of acoustic emanations requires physically implanting a recording device close to the victim, whereas our attack only needs a normal user account. Moreover, these attacks can only be used against a local user. In contrast, our approach works on both local and remote users.

8 Conclusion

In this paper, we present a new attack that allows a malicious user to eavesdrop on other users' keystrokes using procfs, a virtual file system that shares statistic information regarding individual users' processes. Our attack utilizes the stack information of a process present in its `stat` file on a Linux system to fingerprint its behavior when a keystroke is received. Such behavior is modeled as an ESP pattern of its system calls, which can be extracted from an application through automatic program analysis. During the runtime of the application, our approach shadows its process with another process to collect an ESP trace from its `stat` file. Our research shows that on a multi-core system, the shadow process can acquire a trace with a sufficient granularity for identifying keystroke events. This allows us to determine the tim-

ings between keystrokes and analyze them to infer the key sequence the victim entered. We also show that other information available from procfs can be of great help to character inference: knowing that the same user enters her password to the same application, we can combine multiple timing sequences related to the password to significantly reduce the space for searching it. We also propose to utilize the victim's writing style to infer the English words she enters. Both approaches are very effective, according to our experimental study.

Our attack can be further improved through adopting more advanced analysis techniques such as the high-order HMM and other language model. The same idea can also be applied to infer other user activities such as moving and clicking mouse, and even deduce others' secret keys. More generally, other information within procfs, such as system time, can be used for a similar attack, which threatens other UNIX-like systems such as FreeBSD and OpenSolaris. Research in these directions is left as our future work.

Acknowledgements

The authors thank our shepherd Angelos Stavrou for his guidance on the preparation of the final version, and anonymous reviewers for their comments on the draft of the paper. We also thank Rui Wang for his assistance in preparing one of the experiments reported in the paper. This work was supported in part by the National Science Foundation the Cyber Trust program under Grant No. CNS-0716292.

References

[1] Cryptography/frequency analysis. http://en.wikibooks.org/wiki/Cryptography:Frequency_analysis, Aug 2006.

[2] ASONOV, D., AND AGRAWAL, R. Keyboard acoustic emanations. In *IEEE Symposium on Security and Privacy* (2004), pp. 3–11.

[3] BERGER, Y., WOOL, A., AND YEREDOR, A. Dictionary attacks using keyboard acoustics emanations. In *CCS* (2006), ACM, pp. 245–254.

[4] BERGROTH, L., HAKONEN, H., AND RAITA, T. A survey of longest common subsequence algorithms. In *Proceedings of Seventh International Symposium on String Processing and Information Retrieval* (2000), pp. 39–48.

[5] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *In proceedings of the 12th Usenix Security Symposium* (2003).

[6] BUCHHOLTZ, M., GILMORE, S. T., HILLSTON, J., AND NIELSON, F. Securing statically-verified communications protocols against timing attacks. *Electronic Notes in Theoretical Computer Science 128*, 4 (2005), 123–143.

[7] DESIGNER, S., AND SONG, D. Passive analysis of ssh (secure shell) traffic. Openwall advisory OW-003, March 2001.

[8] DHEM, J. F., KOEUNE, F., LEROUX, P.-A., MESTRE, P., QUISQUATER, J.-J., AND WILLEMS, J.-L. A practical implementation of the timing attack. In *Proceedings of CARDIS* (1998), pp. 167–182.

[9] DISTROWATCH.COM. Top ten distributions: An overview of today's top distributions. http://distrowatch.com/dwres.php?resource=major, 2008.

[10] EDIT VIRTUAL LANGUAGE CENTER. Word frequency lists. http://www.edict.com.hk/textanalyser/wordlists.htm, as of September, 2008.

[11] FERRELL, J. procfs: Gone but not forgotten. http://www.freebsd.org/doc/en/articles/linux-users/procfs.html, 2009.

[12] FRANCOIS, M. J., AND PAUL, H. J. Automatic word recognition based on second-order hidden markov models. In *ICSLP* (1994), pp. 247–250.

[13] HOGYE, M. A., HUGHES, C. T., SARFATY, J. M., AND WOLF, J. D. Analysis of the feasibility of keystroke timing attacks over ssh connections. Technical Report CS588, School of Engineering and Applied Science, University of Virginia, December 2001.

[14] INC., R. Process directories. http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/en-US/ReferenceGuide/s2-proc-processdirs.html, 2007.

[15] JONES, N. C., AND PEVZNER, P. A. *An Introduction to Bioinformatics Algorithms*. the MIT Press, August 2004.

[16] JOYCE, R., AND GUPTA, G. Identity authorization based on keystroke latencies. *Communications of the ACM 33*, 2 (1990), 168–176.

[17] KOCHER, P., JAE, J., AND JUN, B. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (1999), Springer-Verlag, pp. 388–397.

[18] LEECH, G., RAYSON, P., AND WILSON, A. Word frequencies in written and spoken english: based on the british national corpus. http://www.comp.lancs.ac.uk/ucrel/bncfreq.

[19] LOSCOCCO, P., AND SMALLEY, S. procfs analysis. http://www.nsa.gov/SeLinux/papers/slinux/node57.html, February 2001.

[20] LUK, C. K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 190–200.

[21] MONROSE, F., AND RUBIN, A. Authentication via keystroke dynamics. In *Proceedings of the 4th ACM conference on Computer and communications security* (1997), ACM Press, pp. 48–56.

[22] PETERSSON, J. What is linux-gate.so.1? http://www.trilithium.com/johan/2005/08/linux-gate/, as of September, 2008.

[23] PROVOS, N. Systrace - interactive policy generation for system calls. http://www.citi.umich.edu/u/provos/systrace/, 2006.

[24] RABINER, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (1989), 257–286.

[25] SECURITY, S. C. Timing analysis is not a real-life threat to ssh secure shell users. http://www.ssh.com/company/news/2001/english/all/article/204/, November 2001.

[26] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium* (2001), USENIX Association.

[27] SOURCEFORGE.NET. `http://sourceforge.net/projects/strace/`, August 2008.

[28] TEAM, G. `http://www.gtk.org`, as of September, 2008.

[29] TEAM, P. Pax address space layout randomization (aslr). `http://pax.grsecurity.net/docs/aslr.txt`, March 2003.

[30] TEAM, P. `http://pax.grsecurity.net/`, as of September, 2008.

[31] TROSTLE, J. Timing attacks against trusted path. In *IEEE Symposium on Security and Privacy* (1998).

[32] TSAFRIR, D., ETSION, Y., AND FEITELSON, D. G. Secretly monopolizing the cpu without superuser privileges. In *Proceedings of 16th USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–18.

[33] ZHANG, L., ZHOU, F., AND TYGAR, J. D. Keyboard acoustic emanations revisited. In *CCS'05: ACM Conference on Computer and Communications Security* (2005), ACM Press, pp. 373–382.

[34] ZHOU, Y., AND FENG, D. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. csrc.nist.gov/groups/STM/cmvp/documents/fips140-3/physec/papers/physecpaper19.pdf, December 2005.

Notes

[1]The program is actually a simplified version of `vim`.

[2]Some old Linux distributions such as RedHat Enterprise 4 do not use vDSO, and instead then entry of their system calls points to `_dl_sysinfo_int80` in library /lib/ld-linux.so or /lib/ld.so.

[3]We designed our attack in a way that a keystroke event can be reliably identified even in the presence of some missing ESP/EIP values, which could happen when the shadow process is preempted by other processes (Section 3).

[4]After the application enter the state that keystroke inputs are expected, our approach waits for a time period before exporting the first sequence. This allows for the accomplishment of all the system calls prior to keystrokes. Similarly, the second sequence is not exported until the keystroke happens for a while so as to ensure that all the system calls related to the stroke are completed.

[5]There are actually two events associated with a keystroke: key press and key release. We use the first event here for the simplicity of explanation. Our technique can actually be applied to both events.

[6]We did not use the instructions such as '`ret`' to identify the end of a call-back function because compiler optimization could remove such instructions from a binary executable.

[7]Some Linux versions such as RedHat [14] turn off the permissions on `maps` but `stat` is always open.

[8]Theoretically, this approach may not eliminate false positives when it comes to non-deterministic applications, because these applications may contain ESP sequences we did not observe during the offline analysis.

[9]The prior work used 10 letters and 5 numbers. We increased the number of letter keys to get a larger set of legitimate words for our experiment on English text.

[10]The factor is actually below what was reported in the prior work [26]. A possibility is that we adopted 225 key pairs rather than 142 used in the prior work.

[11]We did not choose longer words in our experiment to avoid intensive computation. However, such a word can also be learnt through splitting it into shorter segments and analyzing them using different HMMs.

[12]It is reported that FreeBSD moves to phase out procfs [11].

[13]The possibility of timing attack on `SSH` has also been briefly discussed in [26].