The Pennsylvania State University

The Graduate School

SIDE CHANNEL ATTACK FOR KEYSTROKE LOGGING

A Thesis in

Computer Science and Engineering

by

Albert Tannous

© 2008 Albert Tannous

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

May 2008

The thesis of Albert Tannous was reviewed and approved* by the following:

Trent Jaeger

Professor of Computer Science and Engineering

Thesis Advisor

Bhuvan Urgaonkar

Professor of Computer Science and Engineering

Raj Acharya

Department Head and Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

The problem of maintaining the confidentiality of sensitive information in computer systems is typically addressed by mechanisms such as memory protection and access controls to resources. These techniques only protect confidential information through overt channels. Computer hardware and software both can leak sensitive information through covert or side channels. Side channels leak secrets through some observable aspect of a program's execution, such as memory access patterns and power usage. We examine here a side channel created by the X11 window system's translation of keyboard codes to printable character strings. We present an attack method involving timing the translation of keyboard input to printable strings. Finally, we investigate ways in which an unprivileged process can be used to guess passwords via brute force or dictionary, without having subverted the system in any way.

Table of Contents

List of Figures vi

List of Tables vii

Acknowledgments viii

Chapter 1
Introduction 1

Chapter 2
Background 5
2.1 Related Work 5
2.2 Threat Model 7
2.3 X11 Keyboard Input Handling 7
2.4 Linux 2.6 Scheduler 8

Chapter 3
Experimental Setup and Preliminary Experiments 11
3.1 Experimental Setup 11
3.2 Initial Experiment 13
3.2.1 Overview 13
3.2.2 Results 14
3.3 Password Guessing 16
3.3.1 Overview 16
3.3.2 Results 17

Chapter 4
Experiment 1 : Within the Victim Program 19
4.1 Overview 19
4.2 Results 21

Chapter 5
Experiment 2 : Synchronizing the Victim and the Attack Programs 24
5.1 Overview 24

5.2 Results 28

Chapter 6
Experiment 3 : Scheduling the Attack Program **30**
6.1 Overview 30
6.2 Results 32

Chapter 7
Conclusion **36**

Appendix A
XLookupString Code Paths **38**
A.1 KeyBind.c Code 38

Bibliography **48**

List of Figures

1.1 Attack Overview 3

3.1 Initial Experiment 15

4.1 Experiment 1 21

5.1 Experiment 2 27

6.1 Experiment 3 33

List of Tables

2.1 System Calls for the Linux 2.6 Scheduler 10

3.1 The results of timing the keypresses from `trace` 14

4.1 The results of timing the keypresses from `kltest` 22

5.1 The results of timing the keypresses from `trace` using a semaphore 28

6.1 The results of timing the keypresses from a multithreaded `trace` program 34

Acknowledgments

I am most grateful and indebted to my thesis advisor, Dr. Trent Jaeger, for the guidance, patience and encouragement he has shown me during my time here at Penn State. I would also like to thank my other committee member, Dr. Bhuvan Urgaonkar, for his insightful commentary on my work.

Chapter 1

Introduction

One of the fundamental problems of computer security is the protection of the confidentiality of sensitive information in computer systems. The traditional methods of protecting such confidentiality involve imposing restrictions on access to resources which are shared between multiple subjects. Such resources include main memory and non-volatile storage, access to which is controlled by memory protection and access controls respectively. Such traditional methods only protect sensitive information through overt communication channels. That is, they only provide security for channels explicitly defined for the sharing of data.

Along with overt channels, computer systems also contain *covert channels* [1], which were never intended to transmit data, but can be used for doing so anyway. Covert channels have been studied in the context of MLS systems [2], in which they may be used to leak secrets from subjects at high security levels to those at low security levels. A relative of covert channels which is pertinent to a much broader range of systems is the *side channel* [3].

A side channel is any characteristic of a shared resource through which information may be leaked. Normally, side channels are created when the behavior of a program is observably different depending on the data that program is operating on. Thus, some information about that data is leaked through observation of the program's behavior via the side channel. Examples

of side channels include timing channels, through which sensitive information can be leaked by observing execution times, cache channels, where information is leaked based on memory access patterns, power consumption, electromagnetic leaks or even acoustic emanations.

Usually attacks rely on the behavior of specific implementations of cryptographic algorithms, or weaknesses in the algorithm itself to reveal the secret or key. A side channel attack is rather based on information gained from the physical implementation of the system. By observing secrets leaked from privileged processes through these side channels, unprivileged processes can obtain sensitive data without compromising the OS or exploiting bad policy. Thus side channel attacks make it possible to extract secrets with unprivileged processes.

In this work, we focus mainly on a timing attack to recover passwords via keystroke logging. The attack works by observing how long it takes to transfer key information related to the processing of a keyboard input through the timing side channel. To do so, we measure the processing time for each keystroke of the `XLookupString` function, which plays an important part of translating a keystroke to a string. This timing information will help us determine which key was pressed.

Our attack program is an unprivileged process running in user space. The victim program is an X client, also running in user space. Figure 1.1 shows a layout of this attack. When a user enters a key from the X client, it generates a hardware interrupt. The processing then goes through the operating system, the X server, and back to the X client. Our program measures the delay from when the key is entered to when the X client receives the string associated with the key. This timing information, leaked from the X client, allows us to recover the password.

We consider the amount of prerequisite knowledge an attacker must have regarding the target system's behavior, as well as the factors limiting the effectiveness of the attack. We then evaluate feasibility given the aforementioned two constraints. We conclude that the timing attack is a reasonable approach to obtaining passwords, or reducing the password keyspace. Finally, we consider the design and implementation of a program to guess passwords using the system tool

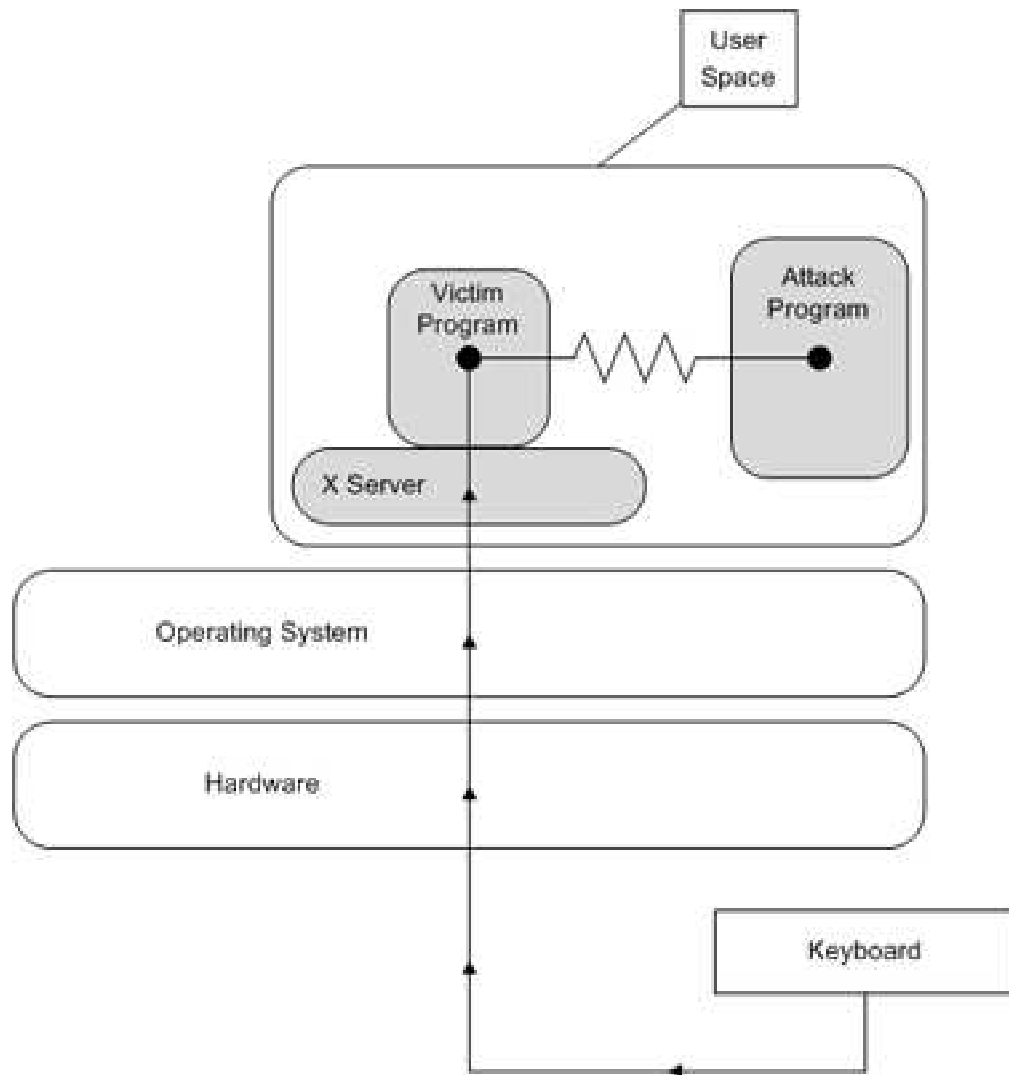

Figure 1.1. Attack Overview

`su` as a password oracle. This method of password guessing was chosen as it requires no special privileges.

This project showed that a timing side channel exists, that the victim program leaks timing information, and that it was possible to get accurate measurements of victim program processing. Using timestamps, we were able to find the keypresses, and even without them, we could significantly reduce the number of possible keypress operations in the output of our attack program.

In the 1st experiment, we claim that we can detect the timing difference of the long XLookup-

String code path when unusual keys are pressed. In the 2nd experiment, we claim that we can detect the timing difference from the attack program if we synchronize it with the victim program. In the 3rd experiment, we make the attack program schedulable to accurately measure the victim's program processing times.

The remainder of this paper is organized as follows. Chapter 2 covers the related work in the area of keylogging and side channel attacks, and explains the necessary background knowledge of the X-Windows system and the Linux 2.6 scheduler. Chapters 3, 4, 5 and 6 describe the experiments we performed to evaluate the side channel attack as well as a method for password guessing by an unprivileged process. We conclude in Chapter 7.

Chapter 2

Background

2.1 Related Work

Keystroke logging refers to any technique by which the keypresses entered at a terminal may be captured and delivered to a third party [4]. Keyloggers manifest themselves in a variety of ways. Some exist as user-space processes which utilize GUI APIs to collect keypresses. Others exist in the kernel, possibly as part of a rootkit, and log all keypresses as keyboard interrupts arrive. These kernel based keyloggers are difficult to discover as their operation is completely transparent to applications. Another class of keylogger relies on hardware between the keyboard and the motherboard [5]. Such external keyloggers are normally inserted between the keyboard and the PS/2 or USB port on the motherboard.

Keystroke loggers must transmit the data they collect to a third party via some means of data exfiltration [6]. Data exfiltration may occur over overt or covert channels. In the overt case, a well known channel such as a network protocol or the physical removal of a storage device is used. In the case of covert channels, data is exfiltrated through a channel not normally used for communication such as disk usage or unused fields in transport level packet headers. One class of keylogger, the keyboard jitterbug [7] exfiltrates keypresses by perturbing the timing of

their transmission. Regardless of the method of exfiltration, the aforementioned methods of keystroke logging rely on an attacker having established some degree of control over the system, whether it be a physical device between the keyboard and computer or malware installed in user or kernel-space.

By utilizing an attack known as a side channel attack [8], we evaluate a method for keylogging which gathers information about passwords without subverting the system. Side channel attacks rely on aspects of a program's execution which give insights into the data the program is acting upon. Some common examples of side channels through which data is leaked are timing information [3], processor caches [9], power usage [10] or even electromagnetic emanations [11]. A process may leak sensitive data through one of these side channels to a spy process monitoring the channel. Cache based attacks normally rely on the spy process timing memory accesses to detect when a victim process has evicted some cache lines [12]. This information combined with knowledge of the algorithm used can give insight into the key used to perform cryptographic operations. Power analysis involves the monitoring of current across wires in a cryptographic device, in order to determine the values being used at various stages of execution.

Virtually all currently known side channel attacks are against cryptographic algorithms. Attacks on the OpenSSL implementation of RSA leverage timing variations in the implementation of modular exponentiation to recover keys [13]. Bernstein [14] describes an attack in which an AES key may be recovered due to the use of parts of the key to index memory locations, allowing for a cache timing attack. Side channel cryptanalysis can even use data implementation to attack product ciphers [15].

Work has been done to investigate the use of side channels to obtain data leaked from the trusted path [16]. Information about passwords can be obtained via a timing attack against the xlock [17] process. Xlock locks the user's screen until the correct password is entered. The timing attack relies on the fact that a keypress event will take longer to process if the user presses a key for which the keysym to string mapping has not yet been loaded into memory. The details

of this mapping are covered in Section 2.3.

Side channels are hard to detect and mitigate and nearly impossible to completely remove [18]. Most side channel countermeasures attempt to introduce noise into channels by randomizing some aspect of program execution or underlying system behavior. These techniques usually only increase the number of measurements needed for the spy process to reconstruct the leaked information. There has been some work with configurable cache architecture to provide hardware assisted defence [19].

2.2 Threat Model

The threat model we are considering is an unprivileged process obtaining entire passwords or parts of passwords without subverting the system. This is done through the use of a side-channel attack which relies on the implementation details of X-windows keyboard input handling [20]. This technique can be used to reduce the keyspace of any user's password including the root user. The timing attack evaluated here deviate from this threat model to varying degrees, the details of which are described in Chapter 3.

2.3 X11 Keyboard Input Handling

We now briefly examine X11's handling of keyboard input [21, 22] for the sake of understanding how these attacks would ideally leverage it. There are three mappings that occur between the time a key is pressed, and the time a character is displayed on the screen in the X Windows system [23]. They are as follows.

1. Physical keys to keycodes: This translation is xserver dependent, and client processes cannot detect this. We will not mention it any further.

2. Keycodes to keysyms: This mapping can be modified by the X clients themselves, but

applies system wide. As we will see, this is a key factor in making the timing attack possible. The keysym is a logical entity which carries the meaning of a keypress. Examples of keysyms include `XK_Return` and `XK_Space`, which represent the return key and the space bar respectively.

3. Keysyms to strings: A keysym itself contains no information about whether or not a character should be displayed for a given keypress or how. It is up to the client process to work with the xserver to perform the keysym to string mapping, where the string is zero or more characters to be printed for a given keypress.

Our timing attack measures the time necessary to complete the `XLookupString` function, which essentially performs phase 3 of the translation. This translation is done via a keysym to string map stored as a linked list.

2.4 Linux 2.6 Scheduler

The new Linux scheduler solved shortcomings of the previous versions and introduces new features [24].

Processes can be classified as I/O bound or processor bound. I/O bound processes spend most of their time submitting and waiting on I/O requests, so they are often runnable, but only for short periods of time. Processor or CPU bound processes execute code most of the time, so they run less frequently, but for longer periods. The scheduler policy in Linux favors I/O bound processes to provide good interactive response.

The Linux scheduler uses a dynamic priority-based scheduling. The processes are ranked based on their worth and need for processor time, but their priority can dynamically change to fulfill scheduling objectives. The Linux kernel implements two separate priority ranges: the nice value, from -20 to 19 with a default of 0, and the real-time priority, from 0 to 99. The nice value is the static priority, because it doesn't change. The decisions of the scheduler are hence based

on the dynamic priority, which is calculated as a function of the static priority and the task's interactivity. To determine whether a process is interactive (I/O bound), the scheduler uses a heuristic based on how long the process sleeps.

The scheduler offers a relatively high default timeslice (100ms), and also dynamically determines the timeslice of a process based on priority. Thus higher priority processes can run longer and more often. When a process enters the TASK_RUNNING state, the kernel checks its priority. If it is higher than the priority of the currently executing process, the scheduler is invoked and picks a new process to run. When the timeslice of a process reaches zero, it is preempted, and another process is selected.

The runqueue is the basic data structure of the scheduler. It's the list of runnable processes on a given processor. Each runqueue contains two priority arrays: the active and the expired array. These arrays contain queues of runnable processes per priority level, as well as a priority bitmap used to find the runnable process with the highest priority. By default, there are 140 priority levels. The active array contains the tasks in the associated runqueue that have timeslice left, while the expired array contains the tasks in the runqueue that have exhausted their timeslice. When the timeslice of a process reaches zero, it's recalculated based on the dynamic priority of the process before the process is moved to the expired array. In addition, if a task is sufficiently interactive, it will be reinserted back into the active array instead of the expired array when it exhausts its timeslice.

A sleeping task is in a non-runnable state. It's removed from the runqueue and put in a wait queue. When the task wakes up, it's runnable again. It's removed from the wake queue and put back in the runqueue.

User preemption is when the kernel is about to return to user-space. It's usually when returning from a system call or an interrupt handler. The scheduler is invoked to continue executing the current task or to pick a new task to execute. The kernel is also preemptive. As long as it is safe to reschedule, a task in the kernel can be preempted at any point. Kernel

System Call	Description
nice()	Set a process's nice value
sched_setscheduler()	Set a process's scheduling policy
sched_getscheduler()	Get a process's scheduling policy
sched_setparam()	Set a process's real-time priority
sched_getparam()	Get a process's real-time priority
sched_get_priority_max()	Get the maximum real-time priority
sched_get_priority_min()	Get the minimum real-time priority
sched_rr_get_interval()	Get a process's timeslice value
sched_setaffinity()	Get a process's processor affinity
sched_getaffinity()	Set a process's processor affinity
sched_yield()	Temporarily yield the processor

Table 2.1. System Calls for the Linux 2.6 Scheduler

preemption can occur when returning to kernel-space from an interrupt handler, or when the kernel code becomes preemtible again. It can also occur explicitly, when a task blocks or calls the scheduler. Linux provides 2 real-time scheduling policies, both of which implement static priorities. The kernel doesn't calculate dynamic priority values for real-time tasks. Hence, a real-time process at a given priority will always preempt a process at a lower priority.

System calls are used for the management of scheduler parameters. A list is showed in Table 2.1.

When a process yields, it is removed from the active array and inserted into the expired array.

Chapter 3

Experimental Setup and Preliminary Experiments

3.1 Experimental Setup

Here are the specifications of the machine used for the experiments.

- Processor: Intel Pentim M-740 (1.73GHz)
- Memory: 2x1GB OCZ DDR2 667 (PC2 5400)
- Hard Disc: Western Digital Scorpio WD1200VE 120GB 5400 RPM ATA-6
- Operating System: Ubuntu 6.10 (Edgy Eft), Linux Kernel version 2.6.17-11

The goal of this timing attack is to use timing information to guess the keys typed by the user. Two programs were used for the experiments. The first one, `kltest`, is used to simulate the X client. Running this program creates a basic X window with a field to type a text. The program waits for a key to be pressed, and then calls the `XLookupString` function - target of our attack - for each character typed. `kltest` also associates a timestamp (using the `gettimeofday()` function) with each keypress. Here are the relevant code sections of the program:

```
while (1) {
```

```
    XNextEvent(display, &report);
    switch (report.type) {
    case KeyPress:

        count = XLookupString(&report, buffer, bufsize, &keysym, &compose);

        gettimeofday(&time1, NULL);
    }
}
```

The second program, `trace`, uses the `rdtsc` [25, 26] instruction to measure the processor's activity. It runs in a tight loop for a predetermined amount of time, continuously taking timestamps and measuring the difference in the number of cycles between each loop iteration. When the program gets context switched out by the scheduler, the cycle difference is significantly higher, indicating that the activity of other processes has been recorded. In such case, the timestamp and the cycle count values are recorded. When the program exits, these values are printed out. Here's the code explaining how it runs:

```
while (time_diff < DURATION) {
    cycles2 = cycles_ia32();
    cycle_diff = cycles2 - cycles1;

    timestamp = gettimeofday(&end,NULL);

    if (cycle_diff > RUN) {
        time_array[i] = timestamp;
        cycle_array[i] = cycle_diff;
        i++;
    }

    cycles1 = cycles2;
}
```

3.2 Initial Experiment

3.2.1 Overview

The first step of the experiment is to remap characters on the keyboard to another character. This means modifying the binding between the keycode and the associated keysym, thus having the keypress resulting in typing the character we chose. By analyzing the source code of `XLookupString`, we noticed that it uses different code paths to process different kinds of characters. This code is shown in Appendix A. We can thus carefully choose to which character we will remap a part of the keyboard, so it will take longer to process these keys.

Here are the basic steps of the experiments:

1. Remap one or more characters
2. Enter a password in the X window
3. Record the processing time for each character
4. Observe the results to determine which keys in the password correspond to a remapped key
5. Repeat until the password is found or the keyspace is significantly reduced.

We decided to remap the chosen characters to the `euro` sign. It's a Unicode character unlikely to be found in a password since it doesn't exist on the `QWERTY` keyboard, and will cause the timing difference we want to exploit. After remapping characters, we launch `kltest` and `trace`, enter the password in `kltest` and take measurements with `trace`. Then we remap other characters on the keyboard, and take other measurements, until we've remapped all the alphanumeric characters. One run of the experiment lasts 10 seconds. We use a shell script to handle the remapping and launch the programs, and Perl scripts [27] to analyze the results.

This experiment has a straightforward approach, as shown in Figure 3.1: run `kltest` and `trace`, enter some characters in `kltest` and take the measurements with `trace`. The first problem we ran into is that keypresses were indistinguishable from the rest of the timing measurements given by `trace`. To identify them, we had to use the timestamps recorded in both `kltest`

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Averages
Duration t	409934	418637	356566	372966	404937	353466	469217	401685	419610.6
Duration e	442933	371930	325033	412428	397869	371348	310033	328947	369507.8
Duration s	324851	292542	307238	324029	353783	368236	437017	284597	340186.1
Duration t	357627	407183	354707	367217	373951	316557	317754	336602	354211.5
Duration €	382847	328927	349502	371340	389249	285492	400560	428232	364253.3
Duration n	434505	373474	358490	456907	387863	335688	376276	388302	401824.8
Duration g	320409	376545	353430	329089	402035	340785	338765	355164	355895.6
Duration 4	402501	315377	352529	354530	378134	284130	346382	394931	361164.3

Table 3.1. The results of timing the keypresses from `trace`

and `trace`. `Kltest` associates a timestamp every time `XLookupString` is called, and `trace` associates one every time it measures a difference in CPU cycles. We used Perl scripts to analyze both outputs and match processing times recorded by `trace` with keypress event times recorded by `kltest`.

3.2.2 Results

The results of that experiment are shown in Table 3.1. As we can see there, the measurements given by `trace` were not precise enough. The durations of all the characters were all in the same range (around 350,000 cycles), and we couldn't distinguish the remapped characters from the non-remapped. That's because the Linux 2.6 scheduler gives a higher priority to I/O bound processes to improve the system's responsiveness and interactivity. Since `trace` runs all the time and exhausts its timeslice every time it's swapped in, the scheduler considers it CPU bound and lowers its priority. Therefore, when `trace` is switched out, the scheduler puts it at the bottom of the runqueue. After `kltest` is done running, `trace` doesn't get switched back in right away. Instead, other processes are allowed to run, like Figure 3.1 shows, and we end up having a lot of noise in our measurements. The behavior of the scheduler is explained more in detail in Section 2.4.

The rest of the experiments will now focus on solving the problems we ran into, to reach a point where we're able to accurately detect a timing difference between remapped and non-

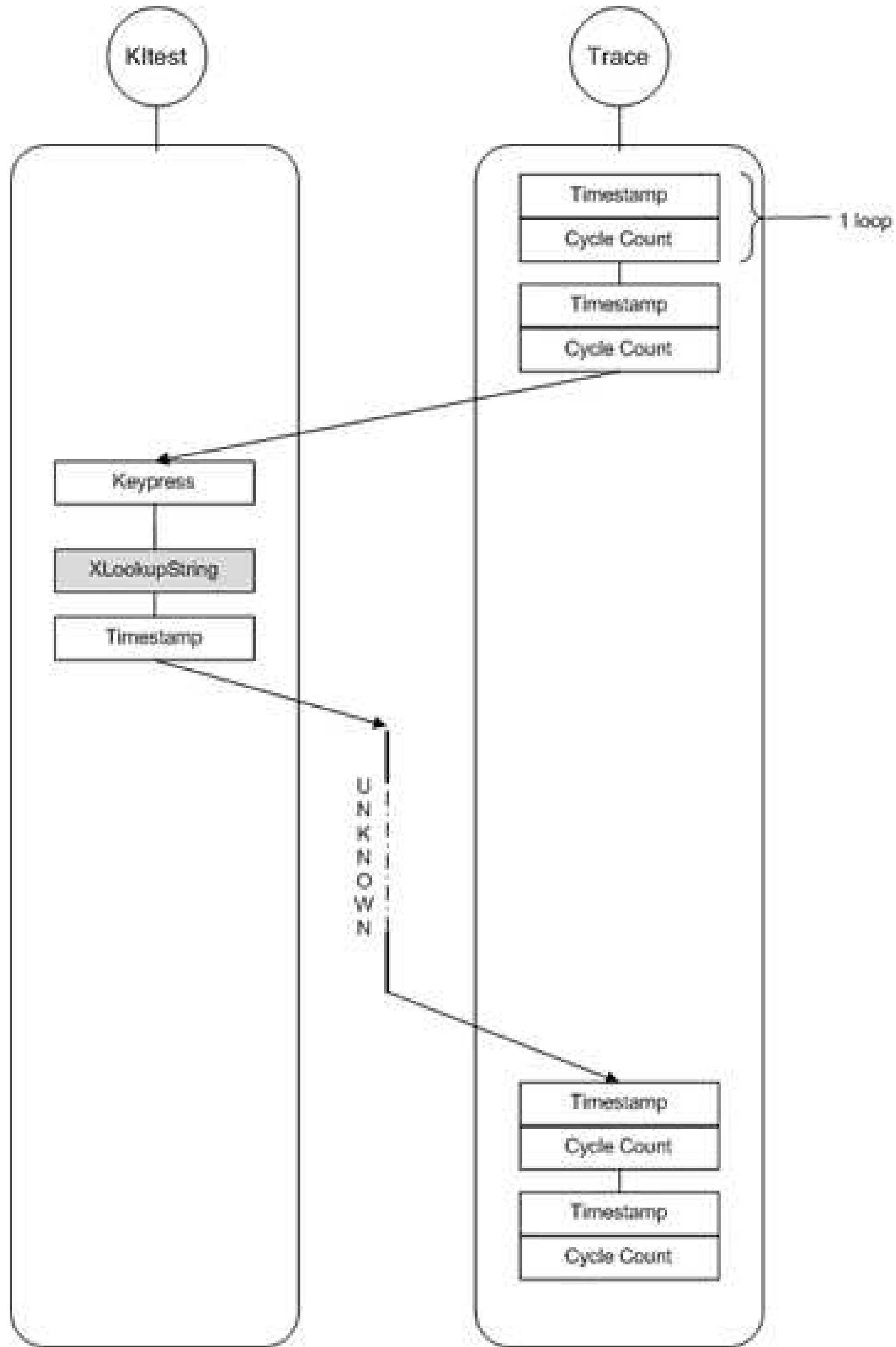

Figure 3.1. Initial Experiment

remapped characters.

Let's note that with the experimental procedure we're following, we can only identify one remapped character per run, and only the first time it's typed (in the case this character is redundant in the input string). After that, the processing time of other remapped characters will be in the same range than the processing time of non remapped characters. The timing difference on the first character is due to the fact that the new code path has to be brought into the cache, so the miss in the L1 instruction cache causes the delay. For the following remapped characters, the code path, and therefore the instructions used will already be in the cache, so they are processed faster. We will confirm that in our next experiment in Chapter 4.

Let's also note that the first character always takes significantly longer to process than the others. That's because when the first character is entered, the keymap is built and brought into the cache. To solve this problem, we sacrificed a character by pressing a random key before starting to input the password in all the experiments.

3.3 Password Guessing

3.3.1 Overview

The attack described here needs several iterations to be able to retrieve a password. Each iteration requires a different character to be mapped, and the user to enter his password. As mentioned in Section 3.2.2, we can detect a remapped character only the first time it's entered, so if a password has a repeated character, we can't detect the second occurrence.

Since the timing attack might not always reveal an entire password, we wanted to investigate methods for guessing passwords given that some characters of the password are known. Because the threat model requires the attacker in the ideal case remain unprivileged, we did not want to use a brute force attack using obtained password hashes such as those stored in `/etc/shadow`. Instead, we chose to use the program `su` as a password oracle, passing it guesses

at the administrator password until we are granted root access.

As it is impractical to attempt to test passwords by hand, we needed to automate the process by writing a password guessing program, `pass`. We encountered several interesting challenges while writing `pass`. First, most implementations programs such as `su`, `sudo`, and `ssh` verify that `stdin`, the file handle for standard input, is associated with a terminal (TTY) device. This prevented us from using simple methods such as:

```
% echo -e "test_password\n" | su
```

to enter passwords. In order to implement `pass` we used Expect [28]. Expect interacts with programs according to a script which specifies the strings to feed to a process inputing from a TTY, and the strings to expect from a process outputting to a TTY. We wrote a Perl script which created many simultaneous instances of Expect, each attempting to send a different password to `su`.

The need to run simultaneous instances was due to another security measure taken by `su` and related programs, the one or two second wait time incurred for an incorrect password. By guessing many passwords in parallel, we significantly mitigated the affects of this wait time by increasing the throughput of password guesses without decreasing the latency for any single guess.

We used the script described above to measure the rate at which we could guess passwords using `su`. This rate may then be used in calculations to find the average time to crack a password of reduced keyspace.

3.3.2 Results

The experiment was run on an Intel Core II Duo processor running Linux kernel 2.6.20-16 with 1 GB RAM. The average rate of password guessing using our script is approximately 90 guesses per second. The average number of instances of Expect running at any given time is approximately 54, with approximately 1,400 additional processes in a “sleeping” state while waiting for `su` to

complete the delay incurred by an incorrect password guess.

To reduce I/O latency due to the writing, executing and deleting of many Expect scripts, the script was run from a ramdisk device, a capability which may or may not be available to an unprivileged process depending on the system. Note that this only minimally increased the guessing rate, suggesting that this method is CPU bound. This is still a very simple approach, and large improvements could be made by implementing the logic of the Perl and Expect scripts in a lower level language such as C. Because of the largely parallel nature of this method, the speedup afforded by additional processors is approximately equal to the number of processor cores on the target system. We verified this by running the script with only one processor core enabled. Under such a restriction, a rate of only approximately 46 guesses per second was achieved, slightly over half of the rate with both processor cores enabled.

At the rate of 90 guesses per second, an eight character UNIX password for which half of the characters were known could be guessed via brute force in approximately ten to eleven days. In the case of an account which is not frequently logged in to, this would be preferable to attempting to remap each character in the keyspace until the full password is recovered. A dictionary constrained by the known characters could further reduce this time.

Chapter 4

Experiment 1 : Within the Victim Program

4.1 Overview

As we saw in 3.2.2, a naive and straightforward approach wasn't successful. In that experiment, `trace` was running all the time, exhausting all its timeslices and behaving like a CPU bound process. The Linux scheduler, designed to favor I/O bound processes over CPU bound ones, as detailed in Section 2.4, would thus lower the priority of `trace`. When a keypress occurs and `kltest` gets swapped in, `trace` enters the runqueue at the bottom, allowing a bunch of processes to run before it runs again. Therefore, the measurements we got had a lot of noise, which made it impossible to detect a timing difference.

However, we feel confident that this difference would be noticeable if measured from within `kltest`. As explained in Section 3.1, `XLookupString` takes a different code path depending on the kind of character entered. Remapping some keys to a Unicode character such as the euro sign '€' would force XLookupString to take a new code path to process it. Since the instructions

of this new path won't be in the L1 instruction cache, that will result in a cache miss, giving us the timing difference that we need.

This timing difference should also be consistent and large enough during the whole password recovery process. To confirm this hypothesis, we tried to recover a password by measuring `XLookupString` directly from `kltest`. The code of `kltest` had to be modified to include the timing information:

```
while (1) {
    XNextEvent(display, &report);
    switch (report.type) {
    case KeyPress:

        cycles_start = cycles_ia32();
        count = XLookupString(&report, buffer, bufsize, &keysym, &compose);
        cycles_end = cycles_ia32();

        cycles_array[j] = cycles_end - cycles_start;
        gettimeofday(&time1, NULL);
        j++;
    }
}
```

This approach would also prevent the scheduler of interfering with our measurements since everything would be happening sequentially, wihin the same process, as Figure 4.1 shows.

To conduct the experiment, we chose an alphanumeric password (that doesn't have any repeated character). The steps were somewhat similar to the ones described in Section 3.2.1:

1. Remap one character
2. Run `kltest`
3. Enter a password in the X window

We used the shell script to remap all the 36 alphanumeric characters, one at a time, to the euro sign. For each character, we entered the chosen password in `kltest` (after hitting a random sacrifice character to build the keymap). At each run, we collected the durations

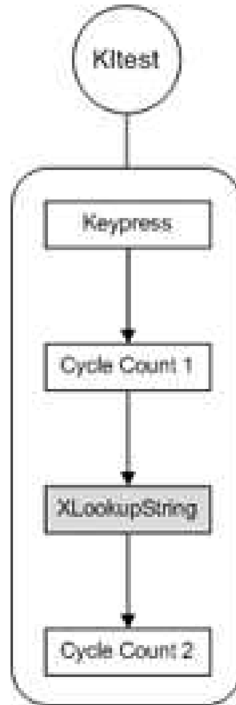

Figure 4.1. Experiment 1

of `XLookupString` for every keypress. After remapping all the 36 characters and entering the password 36 times, we used a Perl script to analyze the results, discussed below.

4.2 Results

This experiment was a success. Table 4.1 summarizes the results. We only display the runs were a remapped character was found. Each column contains the duration of `XLookupString` in CPU cycles for each character of the password “trying57”.

As we can see, we were able to detect the timing difference. It consistently takes over 15,000 additional cycles to process a remapped character. Non-remapped characters are processed in 5,000 to 7,000 cycles, while remapped ones would take over 25,000 cycles. For example, in the column ‘run 5’, the duration of the 7th character is 26,864 which means that this character is remapped. The 5th run corresponds to remapping the alphanumeric character ‘5’. From that we can deduce that the 7th character of the password is ‘5’.

	Run 5	Run 7	Run 14	Run 15	Run 16	Run 18	Run 25	Run 35
Duration t	5560	6865	5920	27794	6927	5110	6649	5635
Duration r	3248	4122	29264	4344	4252	6750	4162	4398
Duration y	4422	4378	4655	3164	37204	5520	6391	5569
Duration i	6413	6051	3537	5930	5492	26498	3525	3971
Duration n	3749	4123	6634	2937	6348	3785	3493	27069
Duration g	5997	5800	4842	3670	5540	6310	29193	4257
Duration 5	26864	3485	6778	6519	6451	4283	3304	3351
Duration 7	4230	26762	5813	3681	4096	3822	5243	3861

Table 4.1. The results of timing the keypresses from `kltest`

Similarly, we were able to find the other 7 characters, and successfully guess the password. For each run (each column), the character that was remapped to the euro sign can easily be determined, as its time is an order of magnitude greater than the others in the same column.

We need at most 36 iterations to get the exact password, but having less than that is also very helpful. Every time the password is entered, additional information is leaked, which reduces the keyspace of the password. This considerably helps a brute force attack, as we discussed in Section 3.3.

To confirm that the timing difference is indeed caused by a miss in the L1 instruction cache, and not by anything else, we expanded this experiment a little. We programmed `kltest` to detect a remapped character as soon as it was entered, so we wouldn't have to wait for the post-processing scripts to analyze the data. If the character entered was a remapped one, we would flush both the L2 cache and the L1 data cache, and then reenter the remapped character. The output of `kltest` showed that we still couldn't identify a second remapped character, which confirmed our previous assumptions. The timing difference was due to a miss in the L1 instruction cache, since the instructions of the different code path of `XLookupString` weren't there.

At this point, our timing attack can discover every letter of a password in a number of attempts equal to the size of the password keyspace by remapping one character at a time. However, it can do this with the extra knowledge of the exact times at which keypress events arrive at the X client, and by measuring their processing times from the X client itself. This shows the next steps

to take in order to reach our final goal. First we need to be able to detect the timing difference from outside the X client. This means we need to be able to get measurements accurate enough from `trace` instead of `kltest`. The main difficulty that arises comes from the Linux scheduler. By lowering the priority of `trace`, it makes it very difficult to get our processes scheduled the way we need them to be. In order to do that, one idea would be to make the scheduler believe that `trace` is an I/O bound process.

Chapter 5

Experiment 2 : Synchronizing the Victim and the Attack Programs

5.1 Overview

We first tried to measure the duration of `XLookupString` from `trace` by running `kltest` and `trace` simultaneously. The full experiment is described in Section 3.2. That naive and straightforward approach didn't work, and we know it's because of the Linux 2.6 scheduler. It makes it impossible to have accurate measurements by simply running `trace` in parallel with `kltest`. The scheduler gives a higher priority to I/O bound processes, and lowers the priority of CPU bound processes. Since `trace` runs during its whole timeslice, it is considered CPU bound by the scheduler. It gets rescheduled at the bottom of the runqueue, and doesn't get to run immediately after `kltest`, thus giving widely inaccurate measurements. The behavior of the scheduler is more thoroughly explained in Section 2.4.

Looking for another approach, we found something we could use in the victim code. By measuring the duration of `XLookupString` directly from `kltest`, our attack was successful. As

we explain in Section 4.2, we were able to detect the timing difference, and even recover the password. We also confirmed that the timing difference is due to a miss in the L1 instruction cache. `XLookupString` takes different code paths to process remapped and non-remapped characters, so the first time a remapped character is entered, the instructions of the new code path have to be brought into the cache. That gives the timing difference.

Now we want to achieve the same results by using `trace` to take the measurements. Based on what we know, we need to work on getting finer grained measurements around `kltest`, and we'll have to do that despite the scheduler's behavior. Since `trace` was being penalized for being CPU bound, we will have the scheduler consider it I/O bound, and give it a higher priority. In order to do that, the program has to spend most of its time sleeping, and wake up at the right moments, for very short periods. We are confident that it would make the measurements recorded by `trace` accurate enough to detect the timing difference, and hence the remapped keys.

A semaphore [29] is a primitive synchronization mechanism, part of the Inter-Process Communication techniques. Using one for this experiment seems to help us do what we want and has several advantages. By having our 2 programs use the same semaphore, we can run `trace` at the exact time we need by triggering it from `kltest`, at the cost of some minor modifications. Here's the modified `kltest` code:

```
while (1) {
    XNextEvent(display, &report);
    switch (report.type) {
    case KeyPress:

        unlock_sem(semid);
        lock_sem(semid);

        cycles_start = cycles_ia32();
        count = XLookupString(&report, buffer, bufsize, &keysym, &compose);
        cycles_end = cycles_ia32();

        cycles_array[j] = cycles_end - cycles_start;
```

```
        unlock_sem(semid1);
        lock_sem(semid1);

        gettimeofday(&time1, NULL);
        j++;
    }
}
```

Since `kltest` will be holding the semaphore while waiting for a keypress, `trace` will be blocked, and will only get switched in when it needs to take the measurements. Even then it will only run for 1 loop, which is a very short period (around 7,000 cycels). This means that it will be sleeping most of the time instead of running. The scheduler will increase its priority, increasing the chances that it gets at the top of the runqueue, and runs right when we want it to. This also required some minor changes in the code of `trace`:

```
while (time_diff < DURATION) {

    lock_sem(semid);

    cycles2 = cycles_ia32();
    cycle_diff = cycles2 - cycles1;

    timestamp = gettimeofday(&end,NULL);

    if (cycle_diff > RUN) {
        time_array[i] = timestamp;
        cycle_array[i] = cycle_diff;
        i++;
    }

    cycles1 = cycles2;

    unlock_sem(semid);
}
```

Using the approach explained above, we were able to schedule `trace` and `kltest` in the order we needed to get good measurements, as Figure 5.1 shows. `Kltest` starts by locking the semaphore, and waits for a keypress. At this time, `trace` is on hold, trying to lock the semaphore.

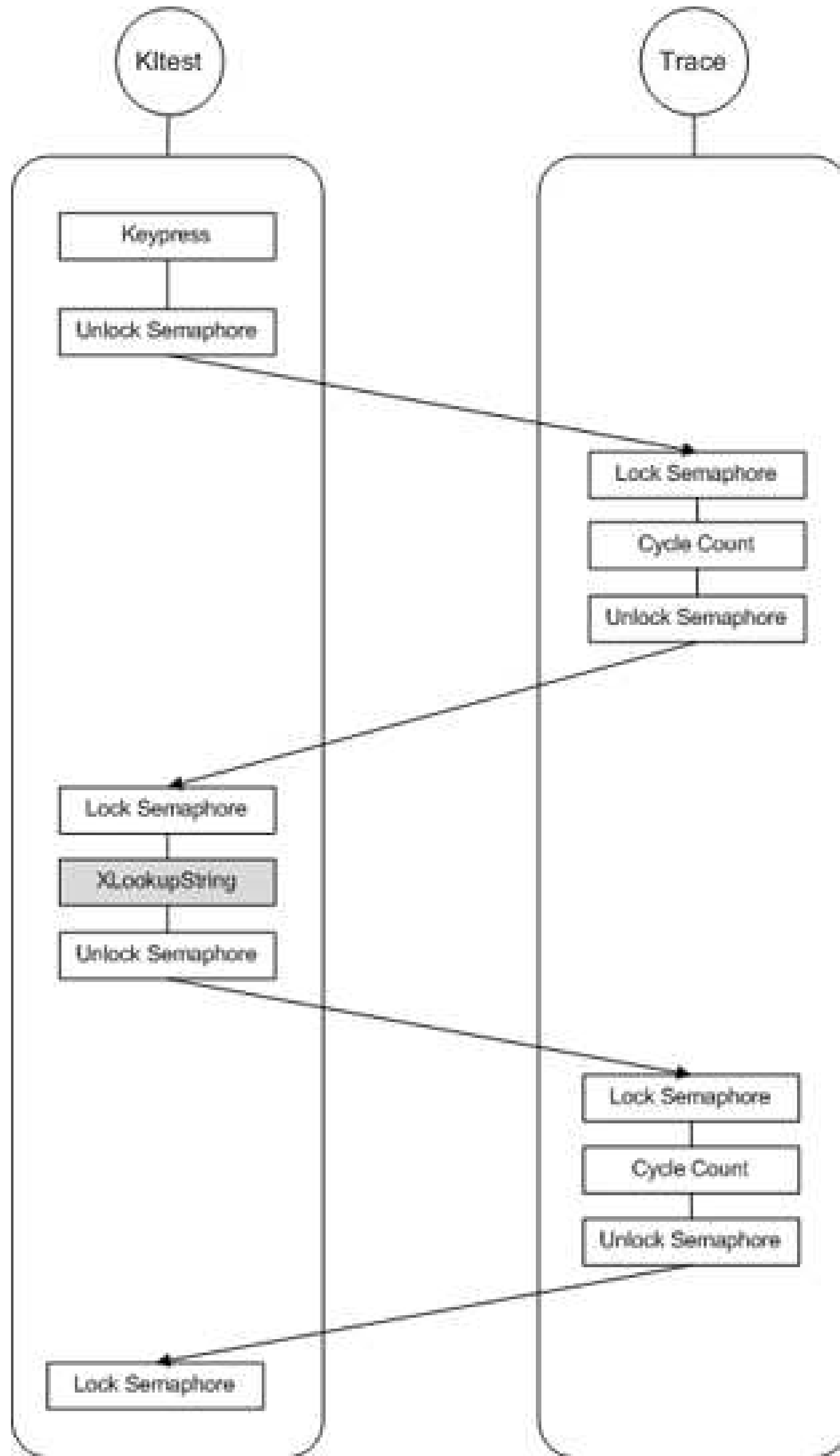

Figure 5.1. Experiment 2

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
duration t	17411	18530	13699	16581	16095	17856	18885	16145
duration r	11917	12123	15643	10424	13821	11791	15281	11400
duration y	11290	14653	10735	14674	12776	16545	11473	15337
duration €	37942	39439	37272	32001	37410	34030	38692	33046
duration f	12901	12288	11615	15361	11746	15822	12345	18327
duration i	12171	16296	15445	11777	17146	11040	14656	12119
duration n	15950	11359	12209	12882	10830	14846	11497	15692
duration d	12563	14813	17275	13609	14462	11441	15997	11042

Table 5.1. The results of timing the keypresses from `trace` using a semaphore

After a key is entered, `kltest` unlocks the semaphore, allowing `trace` to lock it. Trace is allowed to run for 1 loop iteration, takes a measurement, and then unlocks the semaphore, giving it back to `kltest`. `Kltest` locks the semaphore, runs XLookupString, and unlocks the semaphore again. `Trace` locks it, takes another measurement, and unlocks it again, so `kltest` can wait for another keypress. That way, we wrapped the measurements of `trace` around `XLookupString`, since we get it to run just before and just after the victim function.

5.2 Results

By calculating the difference between 2 consecutive measurements of trace, we can measure `XLookupString` duration. The results are summarized in Table 5.1. In this experiment, we entered the string “try2find”, with the character ‘2’ remapped. Here again, we used perl scripts to go through the outputs and match the timestamps between `kltest` and `trace`.

As we can see, we were perfectly able to detect the timing difference. The durations we got here are higher than the durations of `XLookupString` measured from `kltest` in Experiment 1, but they are still consistent. More importantly, the duration of a remapped character is consistently and noticeably higher (by 10,000-15,000 cycles) than the duration of a non-remapped character. The duration of the ‘€’ character is both time above 30,000 cycles, whereas the durations of non-remapped keys are in the 10,000-20,000 cycles range.

The results of this experiment show us that by using a semaphore to prevent `trace` from

running all the time, trigger it from `kltest` and synchronize between the 2, the scheduler would consider `trace` as an I/O bound process. The measurements are fine-grained enough around `XLookupString`. Their precision would allow us to fully recover a password if we remapped all the characters one by one, and entered the password a sufficient number of times, as explained in Chapter 4.

Let's also note that we got similar results by using 2 different semaphores and 2 different `trace` processes running simultaneously. In that case, we used one `trace` to take the first measure and the other one to take the second. Then we calculated the difference between the outputs of both processes. By running 2 different processes, we increased their chances of getting a high priority, since the total run time of `trace` would be divided among them. It wasn't necessary in this experiment, but it's that same reasoning that lead to the next one.

The results we got here strengthen our belief that the scheduler is responsible for the problems we got initially. If we can get it to consider `trace` as I/O bound and increase its priority, we should be able to get consistent measurements and detect the timing difference in `XLookupString`. The challenge now is to be able to have `trace` run all the time to wrap the measurements as close as possible around `kltest`, and still be considered I/O bound by the scheduler. To achieve that, we used multithreading, as our next experiment will show.

Chapter 6

Experiment 3 : Scheduling the Attack Program

6.1 Overview

We've established in Chapter 3 that a straightforward approach didn't work because of the scheduler. In that initial experiment, `trace` was continuously running, so the scheduler considered it CPU bound and reduced its priority. More information about the behavior of the scheduler can be found in Section 2.4. The result of that behavior is that when `trace` would get switched out, a bunch of processes not relevant to our experiment were given a chance to run, and `trace` would end up at the bottom of the runqueue. Therefore the measurements we got were too coarse grain to be able to detect any timing difference between a remapped and a non-remapped character.

We also know that `XLookupString` takes different code paths to process remapped and non-remapped characters. When processing the first remapped character, the instructions of the new code path have to be brought into the cache, and that's what causes the delay we want to detect.

We confirmed that in Chapter 4, when we were able to detect the timing difference by measuring the duration of `XLookupString` directly from `kltest`. In that experiment, we were also able to recover a whole password.

We then tried to detect the timing difference from `trace`. To avoid being handicapped by the scheduler, we allowed `trace` to run for only 1 loop and take a measurement when needed. Because of the fast and precise nature of those measurements, we triggered them from within `kltest`, using a semaphore. That experiment was also successful. Running `trace` at the right time gave measurements precise enough to detect the timing difference and allow us to recover the password.

Our goal now is to be able to schedule `trace` just before and just after keypresses without any trigger or signal from `kltest`. For that we need `trace` to be running all the time, so we can wrap the measurements just around `kltest`, and it has to keep a high enough priority to remain at the top of the runqueue, and not allow other undesirable processes to run in between. If we get the scheduler to give `trace` a high priority by considering it I/O bound, then the measurements taken should be accurate enough to detect remapped characters.

To achieve that, we decided to use multithreading. We launch multiple `trace` threads, each would run for a small portion of its timeslice before being switched out and replaced by another thread. This way, there will always be a `trace` thread running, and another one with a high priority, waiting to replace it at the top of the runqueue. We are sure this approach will allow us to detect the timing difference we need.

We used `pthreads` [30] to conduct this experiment. We launched a large number of `trace` threads, and we allow each one to run for only 1 loop iteration before getting switched out, and replaced by another thread. We used a semaphore to synchronize between all the threads. A thread would lock the semaphore, and unlock it after 1 loop, so another thread can run. That way, all the threads are considered I/O bound, are given a high priority, and at any given time during the experiment, `trace` is running and taking measurements. Here are the relevant parts

of the code used for this experiment:

```
unsigned long long int cycles_global;

void *measure(void *threadid) {
    while (time_diff < DURATION) {
        lock_sem(semid);
        cycles_local = cycles_ia32();
        cycle_diff = cycles_local - cycles_global;

        timestamp = gettimeofday(&end,NULL);

        if (cycle_diff > RUN) {
            measures_array[array_index].thread_id = tid;
            measures_array[array_index].time_measure = timestamp;
            measures_array[array_index].cycle_measure = cycles_local;
            measures_array[array_index].diff_measure = cycle_diff;
            array_index++;
        }

        cycles_global = cycles_local;
        unlock_sem(semid);
    }

int main () {
    for(tc=0;tc<NUM_THREADS;tc++)
        rc = pthread_create(&threads[tc], &attr, measure, (void *)tc);
}
```

In this experiment, all the events are scheduled the way we want, and the measurements from `trace` are nicely wrapped around the `kltest` loop when a keypress in entered. Figure 6.1 shows the sequentiality of the events.

6.2 Results

The outputs of all the threads are recorded in a single array. By using local and global variables, we can calculate the timing difference between a thread and the next. If the switching is fast, it means that no other process ran, and the 2 threads were scheduled consecutively. Each keypress

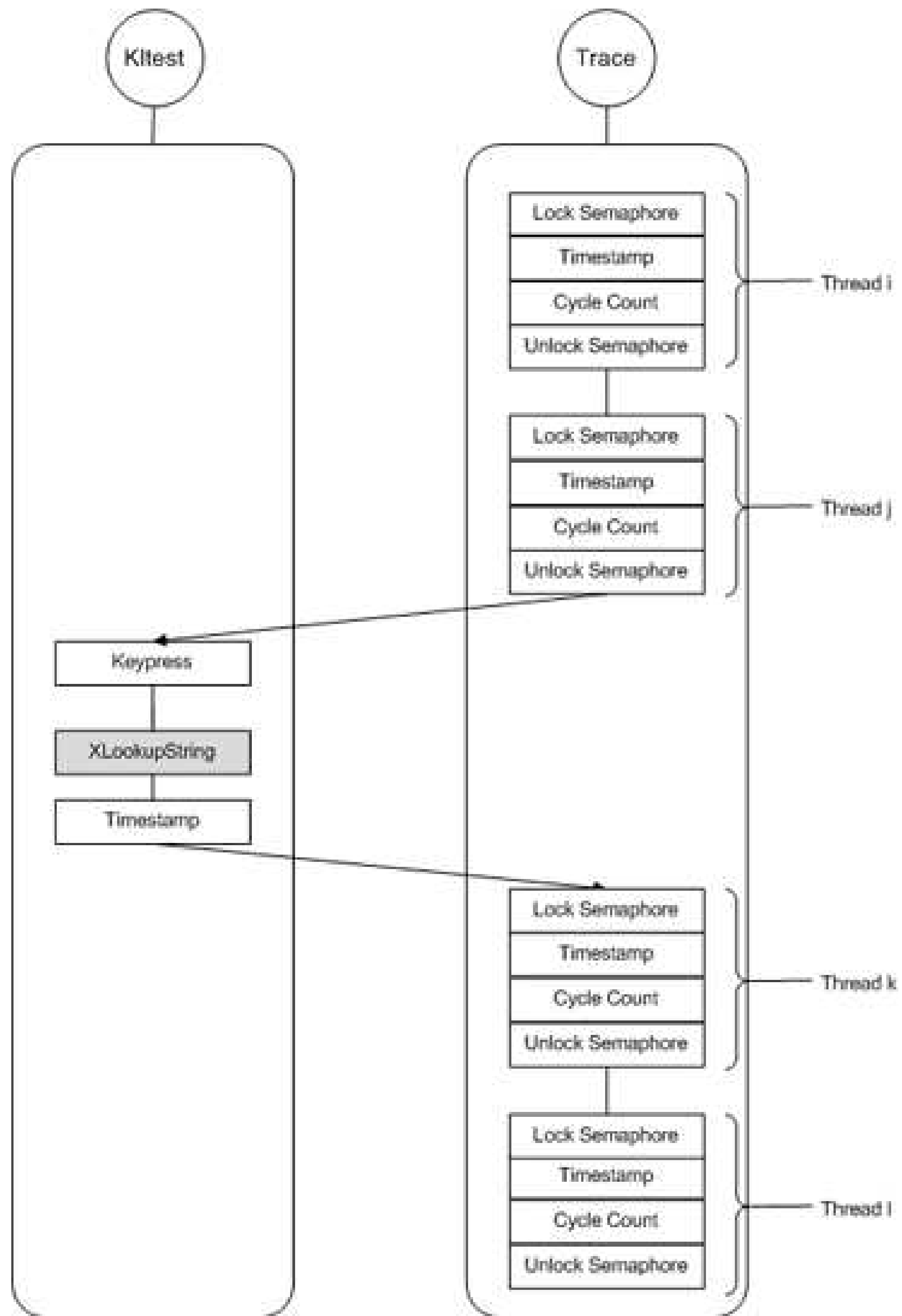

Figure 6.1. Experiment 3

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8
duration t	47147	42468	47223	42488	47572	47920	46662	39489
duration r	31598	33740	32462	30385	43398	42509	42267	26908
duration y	29026	39900	35825	26788	30639	27895	34018	26476
duration €	64316	53888	62909	59987	65099	63281	63402	55778
duration f	31195	39115	32239	35956	28793	35788	27456	26395
duration i	27953	33941	40367	32694	45323	39672	38302	35659
duration n	40314	40005	28643	46316	33041	28648	28388	29220
duration d	27704	32707	28644	29866	43798	43699	41480	27444

Table 6.1. The results of timing the keypresses from a multithreaded `trace` program

corresponds to a timestamp in the output of `kltest`. By looking for the timestamp that comes just after in the output of `trace`, we can find the duration of the processing of that keypress.

This experiment yielded the best results when we launched 50 threads. The results are summarized in Table 6.1. Here again, we entered the string "try2find", with the character '2' remapped. As we can see, the remapped character stands out very clearly. Regular characters are in the range of 30,000 to 45,000 cycles. The line "Duration €" shows that the duration of the remapped characters ranges between 55,000 and 75,000 cycles.

This experiment was successful. The measurements were precise enough to allow us to accurately detect keypresses, and better yet, the timing difference between remapped and non-remapped characters was consistent. These results tell us that we should be able to find the password if we remapped all the characters, and entered the password a sufficient number of times, like we did in Experiment 1, previously explained in Chapter 4.

This confirms that the key to conducting a successful attack is to get the scheduler to consider `trace` as I/O bound and still have our program running the whole duration of the experiment. Doing so gave us accurate measurements and noticeable timing differences between remapped and non-remapped characters, which is what we need.

Let's note that we also tried this experiment, but using multiprocessing instead of multi-threading. We launched multiple `trace` processes, and allowed each one to run for 1 loop only before unlocking the semaphore and getting swapped out. We couldn't find a good number

of processes that would give us accurate measurements. Too few processes weren't enough to run during the whole experiment while keeping a high priority, so other processes were able to run between `kltest` and `trace`. Increasing the number of processes lead to other problems. It increased the overhead of switching processes to a point where even if 2 `trace` processes were wrapped around a `kltest` loop, the measurements we got weren't accurate enough to detect consistent timing difference between remapped and non remapped characters.

At this point, we still need to match either timestamps or cycle counts between `kltest` and `trace` to identify keypresses among the measurements. Our attack can't rely on this information. We can't expect the victim program to let us know when a key was pressed. Future work should focus on detecting keypresses from the output of `trace` alone, without using timestamps. That implies we should be able to tell the difference between the duration of a keypress processing and other processes. The problem here is that a lot of measurements recorded by trace are in the same range as a keypress. We can also try to find an alternative method of detecting a keypress, and combine that information with our output.

Chapter 7

Conclusion

We have presented here a side channel attack method for keystroke logging. The timing based attack is feasible given a modicum of extra information about a system. We showed that knowing the exact times at which keypress events arrived at the X clients, we could determine a password in at most as many tries as there are possible password characters.

We learned here that the Linux 2.6 scheduler I/O bound processes over CPU bound ones, like our attack program. The key to the success of our attack was to fool the scheduler and have it consider our program as I/O bound. That allowed it to keep a high enough priority throughout its execution, and not be interrupted by other processes. To give accurate measurements, our program needs to run just before and just after the victim program, and if its priority is lowered by the scheduler, other processes might run in between. In that case, the attack would fail.

We also created a simple program to test the ability of an unprivileged subject to use common programs such as `su` as a password oracle for the sake of guessing passwords based on the reduced keyspace. We showed that such a program can guess a password for which some characters are known in a reasonable amount of time. So even if we don't get as many tries as our timing attack needs to fully recover the password, the keyspace would have been reduced enough to make a brute force attack possible, and we can still find that password.

Our timing attack still needs timestamps to be successful. A real life victim client won't provide us with such information. Future work should focus on detecting keypresses without timestamps. The fact that a keypress doesn't have an uniquely identifiable processing time makes that challenging.

Appendix A

XLookupString Code Paths

A.1 KeyBind.c Code

```
static void
UCSConvertCase( register unsigned code,
                KeySym *lower,
                KeySym *upper )
{
    /* Case conversion for UCS, as in Unicode Data version 4.0.0 */
    /* NB: Only converts simple one-to-one mappings. */

    /* Tables are used where they take less space than     */
    /* the code to work out the mappings. Zero values mean */
    /* undefined code points.                              */

    static unsigned short const IPAExt_upper_mapping[] = { /* part only */
                            0x0181, 0x0186, 0x0255, 0x0189, 0x018A,
    0x0258, 0x018F, 0x025A, 0x0190, 0x025C, 0x025D, 0x025E, 0x025F,
    0x0193, 0x0261, 0x0262, 0x0194, 0x0264, 0x0265, 0x0266, 0x0267,
    0x0197, 0x0196, 0x026A, 0x026B, 0x026C, 0x026D, 0x026E, 0x019C,
    0x0270, 0x0271, 0x019D, 0x0273, 0x0274, 0x019F, 0x0276, 0x0277,
    0x0278, 0x0279, 0x027A, 0x027B, 0x027C, 0x027D, 0x027E, 0x027F,
    0x01A6, 0x0281, 0x0282, 0x01A9, 0x0284, 0x0285, 0x0286, 0x0287,
    0x01AE, 0x0289, 0x01B1, 0x01B2, 0x028C, 0x028D, 0x028E, 0x028F,
    0x0290, 0x0291, 0x01B7
    };

    static unsigned short const LatinExtB_upper_mapping[] = { /* first part only */
    0x0180, 0x0181, 0x0182, 0x0182, 0x0184, 0x0184, 0x0186, 0x0187,
```

```
0x0187, 0x0189, 0x018A, 0x018B, 0x018B, 0x018D, 0x018E, 0x018F,
0x0190, 0x0191, 0x0191, 0x0193, 0x0194, 0x01F6, 0x0196, 0x0197,
0x0198, 0x0198, 0x019A, 0x019B, 0x019C, 0x019D, 0x0220, 0x019F,
0x01A0, 0x01A0, 0x01A2, 0x01A2, 0x01A4, 0x01A4, 0x01A6, 0x01A7,
0x01A7, 0x01A9, 0x01AA, 0x01AB, 0x01AC, 0x01AC, 0x01AE, 0x01AF,
0x01AF, 0x01B1, 0x01B2, 0x01B3, 0x01B3, 0x01B5, 0x01B5, 0x01B7,
0x01B8, 0x01B8, 0x01BA, 0x01BB, 0x01BC, 0x01BC, 0x01BE, 0x01F7,
0x01C0, 0x01C1, 0x01C2, 0x01C3, 0x01C4, 0x01C4, 0x01C4, 0x01C7,
0x01C7, 0x01C7, 0x01CA, 0x01CA, 0x01CA
};

static unsigned short const LatinExtB_lower_mapping[] = { /* first part only */
0x0180, 0x0253, 0x0183, 0x0183, 0x0185, 0x0185, 0x0254, 0x0188,
0x0188, 0x0256, 0x0257, 0x018C, 0x018C, 0x018D, 0x01DD, 0x0259,
0x025B, 0x0192, 0x0192, 0x0260, 0x0263, 0x0195, 0x0269, 0x0268,
0x0199, 0x0199, 0x019A, 0x019B, 0x026F, 0x0272, 0x019E, 0x0275,
0x01A1, 0x01A1, 0x01A3, 0x01A3, 0x01A5, 0x01A5, 0x0280, 0x01A8,
0x01A8, 0x0283, 0x01AA, 0x01AB, 0x01AD, 0x01AD, 0x0288, 0x01B0,
0x01B0, 0x028A, 0x028B, 0x01B4, 0x01B4, 0x01B6, 0x01B6, 0x0292,
0x01B9, 0x01B9, 0x01BA, 0x01BB, 0x01BD, 0x01BD, 0x01BE, 0x01BF,
0x01C0, 0x01C1, 0x01C2, 0x01C3, 0x01C6, 0x01C6, 0x01C6, 0x01C9,
0x01C9, 0x01C9, 0x01CC, 0x01CC, 0x01CC
};

static unsigned short const Greek_upper_mapping[] = {
0x0000, 0x0000, 0x0000, 0x0000, 0x0374, 0x0375, 0x0000, 0x0000,
0x0000, 0x0000, 0x037A, 0x0000, 0x0000, 0x0000, 0x037E, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0384, 0x0385, 0x0386, 0x0387,
0x0388, 0x0389, 0x038A, 0x0000, 0x038C, 0x0000, 0x038E, 0x038F,
0x0390, 0x0391, 0x0392, 0x0393, 0x0394, 0x0395, 0x0396, 0x0397,
0x0398, 0x0399, 0x039A, 0x039B, 0x039C, 0x039D, 0x039E, 0x039F,
0x03A0, 0x03A1, 0x0000, 0x03A3, 0x03A4, 0x03A5, 0x03A6, 0x03A7,
0x03A8, 0x03A9, 0x03AA, 0x03AB, 0x0386, 0x0388, 0x0389, 0x038A,
0x03B0, 0x0391, 0x0392, 0x0393, 0x0394, 0x0395, 0x0396, 0x0397,
0x0398, 0x0399, 0x039A, 0x039B, 0x039C, 0x039D, 0x039E, 0x039F,
0x03A0, 0x03A1, 0x03A3, 0x03A3, 0x03A4, 0x03A5, 0x03A6, 0x03A7,
0x03A8, 0x03A9, 0x03AA, 0x03AB, 0x038C, 0x038E, 0x038F, 0x0000,
0x0392, 0x0398, 0x03D2, 0x03D3, 0x03D4, 0x03A6, 0x03A0, 0x03D7,
0x03D8, 0x03D8, 0x03DA, 0x03DA, 0x03DC, 0x03DC, 0x03DE, 0x03DE,
0x03E0, 0x03E0, 0x03E2, 0x03E2, 0x03E4, 0x03E4, 0x03E6, 0x03E6,
0x03E8, 0x03E8, 0x03EA, 0x03EA, 0x03EC, 0x03EC, 0x03EE, 0x03EE,
0x039A, 0x03A1, 0x03F9, 0x03F3, 0x03F4, 0x0395, 0x03F6, 0x03F7,
0x03F7, 0x03F9, 0x03FA, 0x03FA, 0x0000, 0x0000, 0x0000, 0x0000
};

static unsigned short const Greek_lower_mapping[] = {
0x0000, 0x0000, 0x0000, 0x0000, 0x0374, 0x0375, 0x0000, 0x0000,
0x0000, 0x0000, 0x037A, 0x0000, 0x0000, 0x0000, 0x037E, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0384, 0x0385, 0x03AC, 0x0387,
0x03AD, 0x03AE, 0x03AF, 0x0000, 0x03CC, 0x0000, 0x03CD, 0x03CE,
0x0390, 0x03B1, 0x03B2, 0x03B3, 0x03B4, 0x03B5, 0x03B6, 0x03B7,
```

```
0x03B8, 0x03B9, 0x03BA, 0x03BB, 0x03BC, 0x03BD, 0x03BE, 0x03BF,
0x03C0, 0x03C1, 0x0000, 0x03C3, 0x03C4, 0x03C5, 0x03C6, 0x03C7,
0x03C8, 0x03C9, 0x03CA, 0x03CB, 0x03AC, 0x03AD, 0x03AE, 0x03AF,
0x03B0, 0x03B1, 0x03B2, 0x03B3, 0x03B4, 0x03B5, 0x03B6, 0x03B7,
0x03B8, 0x03B9, 0x03BA, 0x03BB, 0x03BC, 0x03BD, 0x03BE, 0x03BF,
0x03C0, 0x03C1, 0x03C2, 0x03C3, 0x03C4, 0x03C5, 0x03C6, 0x03C7,
0x03C8, 0x03C9, 0x03CA, 0x03CB, 0x03CC, 0x03CD, 0x03CE, 0x0000,
0x03D0, 0x03D1, 0x03D2, 0x03D3, 0x03D4, 0x03D5, 0x03D6, 0x03D7,
0x03D9, 0x03D9, 0x03DB, 0x03DB, 0x03DD, 0x03DD, 0x03DF, 0x03DF,
0x03E1, 0x03E1, 0x03E3, 0x03E3, 0x03E5, 0x03E5, 0x03E7, 0x03E7,
0x03E9, 0x03E9, 0x03EB, 0x03EB, 0x03ED, 0x03ED, 0x03EF, 0x03EF,
0x03F0, 0x03F1, 0x03F2, 0x03F3, 0x03B8, 0x03F5, 0x03F6, 0x03F8,
0x03F8, 0x03F2, 0x03FB, 0x03FB, 0x0000, 0x0000, 0x0000, 0x0000
};

static unsigned short const GreekExt_lower_mapping[] = {
0x1F00, 0x1F01, 0x1F02, 0x1F03, 0x1F04, 0x1F05, 0x1F06, 0x1F07,
0x1F00, 0x1F01, 0x1F02, 0x1F03, 0x1F04, 0x1F05, 0x1F06, 0x1F07,
0x1F10, 0x1F11, 0x1F12, 0x1F13, 0x1F14, 0x1F15, 0x0000, 0x0000,
0x1F10, 0x1F11, 0x1F12, 0x1F13, 0x1F14, 0x1F15, 0x0000, 0x0000,
0x1F20, 0x1F21, 0x1F22, 0x1F23, 0x1F24, 0x1F25, 0x1F26, 0x1F27,
0x1F20, 0x1F21, 0x1F22, 0x1F23, 0x1F24, 0x1F25, 0x1F26, 0x1F27,
0x1F30, 0x1F31, 0x1F32, 0x1F33, 0x1F34, 0x1F35, 0x1F36, 0x1F37,
0x1F30, 0x1F31, 0x1F32, 0x1F33, 0x1F34, 0x1F35, 0x1F36, 0x1F37,
0x1F40, 0x1F41, 0x1F42, 0x1F43, 0x1F44, 0x1F45, 0x0000, 0x0000,
0x1F40, 0x1F41, 0x1F42, 0x1F43, 0x1F44, 0x1F45, 0x0000, 0x0000,
0x1F50, 0x1F51, 0x1F52, 0x1F53, 0x1F54, 0x1F55, 0x1F56, 0x1F57,
0x0000, 0x1F51, 0x0000, 0x1F53, 0x0000, 0x1F55, 0x0000, 0x1F57,
0x1F60, 0x1F61, 0x1F62, 0x1F63, 0x1F64, 0x1F65, 0x1F66, 0x1F67,
0x1F60, 0x1F61, 0x1F62, 0x1F63, 0x1F64, 0x1F65, 0x1F66, 0x1F67,
0x1F70, 0x1F71, 0x1F72, 0x1F73, 0x1F74, 0x1F75, 0x1F76, 0x1F77,
0x1F78, 0x1F79, 0x1F7A, 0x1F7B, 0x1F7C, 0x1F7D, 0x0000, 0x0000,
0x1F80, 0x1F81, 0x1F82, 0x1F83, 0x1F84, 0x1F85, 0x1F86, 0x1F87,
0x1F80, 0x1F81, 0x1F82, 0x1F83, 0x1F84, 0x1F85, 0x1F86, 0x1F87,
0x1F90, 0x1F91, 0x1F92, 0x1F93, 0x1F94, 0x1F95, 0x1F96, 0x1F97,
0x1F90, 0x1F91, 0x1F92, 0x1F93, 0x1F94, 0x1F95, 0x1F96, 0x1F97,
0x1FA0, 0x1FA1, 0x1FA2, 0x1FA3, 0x1FA4, 0x1FA5, 0x1FA6, 0x1FA7,
0x1FA0, 0x1FA1, 0x1FA2, 0x1FA3, 0x1FA4, 0x1FA5, 0x1FA6, 0x1FA7,
0x1FB0, 0x1FB1, 0x1FB2, 0x1FB3, 0x1FB4, 0x0000, 0x1FB6, 0x1FB7,
0x1FB0, 0x1FB1, 0x1F70, 0x1F71, 0x1FB3, 0x1FBD, 0x1FBE, 0x1FBF,
0x1FC0, 0x1FC1, 0x1FC2, 0x1FC3, 0x1FC4, 0x0000, 0x1FC6, 0x1FC7,
0x1F72, 0x1F73, 0x1F74, 0x1F75, 0x1FC3, 0x1FCD, 0x1FCE, 0x1FCF,
0x1FD0, 0x1FD1, 0x1FD2, 0x1FD3, 0x0000, 0x0000, 0x1FD6, 0x1FD7,
0x1FD0, 0x1FD1, 0x1F76, 0x1F77, 0x0000, 0x1FDD, 0x1FDE, 0x1FDF,
0x1FE0, 0x1FE1, 0x1FE2, 0x1FE3, 0x1FE4, 0x1FE5, 0x1FE6, 0x1FE7,
0x1FE0, 0x1FE1, 0x1F7A, 0x1F7B, 0x1FE5, 0x1FED, 0x1FEE, 0x1FEF,
0x0000, 0x0000, 0x1FF2, 0x1FF3, 0x1FF4, 0x0000, 0x1FF6, 0x1FF7,
0x1F78, 0x1F79, 0x1F7C, 0x1F7D, 0x1FF3, 0x1FFD, 0x1FFE, 0x0000
};

static unsigned short const GreekExt_upper_mapping[] = {
```

```
0x1F08, 0x1F09, 0x1F0A, 0x1F0B, 0x1F0C, 0x1F0D, 0x1F0E, 0x1F0F,
0x1F08, 0x1F09, 0x1F0A, 0x1F0B, 0x1F0C, 0x1F0D, 0x1F0E, 0x1F0F,
0x1F18, 0x1F19, 0x1F1A, 0x1F1B, 0x1F1C, 0x1F1D, 0x0000, 0x0000,
0x1F18, 0x1F19, 0x1F1A, 0x1F1B, 0x1F1C, 0x1F1D, 0x0000, 0x0000,
0x1F28, 0x1F29, 0x1F2A, 0x1F2B, 0x1F2C, 0x1F2D, 0x1F2E, 0x1F2F,
0x1F28, 0x1F29, 0x1F2A, 0x1F2B, 0x1F2C, 0x1F2D, 0x1F2E, 0x1F2F,
0x1F38, 0x1F39, 0x1F3A, 0x1F3B, 0x1F3C, 0x1F3D, 0x1F3E, 0x1F3F,
0x1F38, 0x1F39, 0x1F3A, 0x1F3B, 0x1F3C, 0x1F3D, 0x1F3E, 0x1F3F,
0x1F48, 0x1F49, 0x1F4A, 0x1F4B, 0x1F4C, 0x1F4D, 0x0000, 0x0000,
0x1F48, 0x1F49, 0x1F4A, 0x1F4B, 0x1F4C, 0x1F4D, 0x0000, 0x0000,
0x1F50, 0x1F59, 0x1F52, 0x1F5B, 0x1F54, 0x1F5D, 0x1F56, 0x1F5F,
0x0000, 0x1F59, 0x0000, 0x1F5B, 0x0000, 0x1F5D, 0x0000, 0x1F5F,
0x1F68, 0x1F69, 0x1F6A, 0x1F6B, 0x1F6C, 0x1F6D, 0x1F6E, 0x1F6F,
0x1F68, 0x1F69, 0x1F6A, 0x1F6B, 0x1F6C, 0x1F6D, 0x1F6E, 0x1F6F,
0x1FBA, 0x1FBB, 0x1FC8, 0x1FC9, 0x1FCA, 0x1FCB, 0x1FDA, 0x1FDB,
0x1FF8, 0x1FF9, 0x1FEA, 0x1FEB, 0x1FFA, 0x1FFB, 0x0000, 0x0000,
0x1F88, 0x1F89, 0x1F8A, 0x1F8B, 0x1F8C, 0x1F8D, 0x1F8E, 0x1F8F,
0x1F88, 0x1F89, 0x1F8A, 0x1F8B, 0x1F8C, 0x1F8D, 0x1F8E, 0x1F8F,
0x1F98, 0x1F99, 0x1F9A, 0x1F9B, 0x1F9C, 0x1F9D, 0x1F9E, 0x1F9F,
0x1F98, 0x1F99, 0x1F9A, 0x1F9B, 0x1F9C, 0x1F9D, 0x1F9E, 0x1F9F,
0x1FA8, 0x1FA9, 0x1FAA, 0x1FAB, 0x1FAC, 0x1FAD, 0x1FAE, 0x1FAF,
0x1FA8, 0x1FA9, 0x1FAA, 0x1FAB, 0x1FAC, 0x1FAD, 0x1FAE, 0x1FAF,
0x1FB8, 0x1FB9, 0x1FB2, 0x1FBC, 0x1FB4, 0x0000, 0x1FB6, 0x1FB7,
0x1FB8, 0x1FB9, 0x1FBA, 0x1FBB, 0x1FBC, 0x1FBD, 0x0399, 0x1FBF,
0x1FC0, 0x1FC1, 0x1FC2, 0x1FCC, 0x1FC4, 0x0000, 0x1FC6, 0x1FC7,
0x1FC8, 0x1FC9, 0x1FCA, 0x1FCB, 0x1FCC, 0x1FCD, 0x1FCE, 0x1FCF,
0x1FD8, 0x1FD9, 0x1FD2, 0x1FD3, 0x0000, 0x0000, 0x1FD6, 0x1FD7,
0x1FD8, 0x1FD9, 0x1FDA, 0x1FDB, 0x0000, 0x1FDD, 0x1FDE, 0x1FDF,
0x1FE8, 0x1FE9, 0x1FE2, 0x1FE3, 0x1FE4, 0x1FEC, 0x1FE6, 0x1FE7,
0x1FE8, 0x1FE9, 0x1FEA, 0x1FEB, 0x1FEC, 0x1FED, 0x1FEE, 0x1FEF,
0x0000, 0x0000, 0x1FF2, 0x1FFC, 0x1FF4, 0x0000, 0x1FF6, 0x1FF7,
0x1FF8, 0x1FF9, 0x1FFA, 0x1FFB, 0x1FFC, 0x1FFD, 0x1FFE, 0x0000
};

*lower = code;
*upper = code;

/* Basic Latin and Latin-1 Supplement, U+0000 to U+00FF */
if (code <= 0x00ff) {
    if (code >= 0x0041 && code <= 0x005a)             /* A-Z */
        *lower += 0x20;
    else if (code >= 0x0061 && code <= 0x007a)        /* a-z */
        *upper -= 0x20;
    else if ( (code >= 0x00c0 && code <= 0x00d6) ||
      (code >= 0x00d8 && code <= 0x00de) )
        *lower += 0x20;
    else if ( (code >= 0x00e0 && code <= 0x00f6) ||
      (code >= 0x00f8 && code <= 0x00fe) )
        *upper -= 0x20;
    else if (code == 0x00ff)      /* y with diaeresis */
        *upper = 0x0178;
```

```
        else if (code == 0x00b5)      /* micro sign */
            *upper = 0x039c;
return;
    }

    /* Latin Extended-A, U+0100 to U+017F */
    if (code >= 0x0100 && code <= 0x017f) {
        if ( (code >= 0x0100 && code <= 0x012f) ||
             (code >= 0x0132 && code <= 0x0137) ||
             (code >= 0x014a && code <= 0x0177) ) {
            *upper = code & ~1;
            *lower = code | 1;
        }
        else if ( (code >= 0x0139 && code <= 0x0148) ||
                  (code >= 0x0179 && code <= 0x017e) ) {
            if (code & 1)
        *lower += 1;
            else
        *upper -= 1;
        }
        else if (code == 0x0130)
            *lower = 0x0069;
        else if (code == 0x0131)
            *upper = 0x0049;
        else if (code == 0x0178)
            *lower = 0x00ff;
        else if (code == 0x017f)
            *upper = 0x0053;
        return;
    }

    /* Latin Extended-B, U+0180 to U+024F */
    if (code >= 0x0180 && code <= 0x024f) {
        if (code >= 0x01cd && code <= 0x01dc) {
    if (code & 1)
       *lower += 1;
    else
       *upper -= 1;
        }
        else if ( (code >= 0x01de && code <= 0x01ef) ||
                  (code >= 0x01f4 && code <= 0x01f5) ||
                  (code >= 0x01f8 && code <= 0x021f) ||
                  (code >= 0x0222 && code <= 0x0233) ) {
            *lower |= 1;
            *upper &= ~1;
        }
        else if (code >= 0x0180 && code <= 0x01cc) {
            *lower = LatinExtB_lower_mapping[code - 0x0180];
            *upper = LatinExtB_upper_mapping[code - 0x0180];
        }
        else if (code == 0x01dd)
```

```
        *upper = 0x018e;
    else if (code == 0x01f1 || code == 0x01f2) {
        *lower = 0x01f3;
        *upper = 0x01f1;
    }
    else if (code == 0x01f3)
        *upper = 0x01f1;
    else if (code == 0x01f6)
        *lower = 0x0195;
    else if (code == 0x01f7)
        *lower = 0x01bf;
    else if (code == 0x0220)
        *lower = 0x019e;
    return;
}

/* IPA Extensions, U+0250 to U+02AF */
if (code >= 0x0253 && code <= 0x0292) {
    *upper = IPAExt_upper_mapping[code - 0x0253];
}

/* Combining Diacritical Marks, U+0300 to U+036F */
if (code == 0x0345) {
    *upper = 0x0399;
}

/* Greek and Coptic, U+0370 to U+03FF */
if (code >= 0x0370 && code <= 0x03ff) {
    *lower = Greek_lower_mapping[code - 0x0370];
    *upper = Greek_upper_mapping[code - 0x0370];
    if (*upper == 0)
        *upper = code;
    if (*lower == 0)
        *lower = code;
}

/* Cyrillic and Cyrillic Supplementary, U+0400 to U+052F */
if ( (code >= 0x0400 && code <= 0x04ff) ||
     (code >= 0x0500 && code <= 0x052f) ) {
    if (code >= 0x0400 && code <= 0x040f)
        *lower += 0x50;
    else if (code >= 0x0410 && code <= 0x042f)
        *lower += 0x20;
    else if (code >= 0x0430 && code <= 0x044f)
        *upper -= 0x20;
    else if (code >= 0x0450 && code <= 0x045f)
        *upper -= 0x50;
    else if ( (code >= 0x0460 && code <= 0x0481) ||
              (code >= 0x048a && code <= 0x04bf) ||
      (code >= 0x04d0 && code <= 0x04f5) ||
      (code >= 0x04f8 && code <= 0x04f9) ||
```

```
            (code >= 0x0500 && code <= 0x050f) ) {
        *upper &= ~1;
        *lower |= 1;
    }
    else if (code >= 0x04c1 && code <= 0x04ce) {
if (code & 1)
    *lower += 1;
else
    *upper -= 1;
    }
}

/* Armenian, U+0530 to U+058F */
if (code >= 0x0530 && code <= 0x058f) {
    if (code >= 0x0531 && code <= 0x0556)
        *lower += 0x30;
    else if (code >=0x0561 && code <= 0x0586)
        *upper -= 0x30;
}

/* Latin Extended Additional, U+1E00 to U+1EFF */
if (code >= 0x1e00 && code <= 0x1eff) {
    if ( (code >= 0x1e00 && code <= 0x1e95) ||
         (code >= 0x1ea0 && code <= 0x1ef9) ) {
        *upper &= ~1;
        *lower |= 1;
    }
    else if (code == 0x1e9b)
        *upper = 0x1e60;
}

/* Greek Extended, U+1F00 to U+1FFF */
if (code >= 0x1f00 && code <= 0x1fff) {
    *lower = GreekExt_lower_mapping[code - 0x1f00];
    *upper = GreekExt_upper_mapping[code - 0x1f00];
    if (*upper == 0)
        *upper = code;
    if (*lower == 0)
        *lower = code;
}

/* Letterlike Symbols, U+2100 to U+214F */
if (code >= 0x2100 && code <= 0x214f) {
    switch (code) {
    case 0x2126: *lower = 0x03c9; break;
    case 0x212a: *lower = 0x006b; break;
    case 0x212b: *lower = 0x00e5; break;
    }
}
/* Number Forms, U+2150 to U+218F */
else if (code >= 0x2160 && code <= 0x216f)
```

```
        *lower += 0x10;
    else if (code >= 0x2170 && code <= 0x217f)
        *upper -= 0x10;
    /* Enclosed Alphanumerics, U+2460 to U+24FF */
    else if (code >= 0x24b6 && code <= 0x24cf)
        *lower += 0x1a;
    else if (code >= 0x24d0 && code <= 0x24e9)
        *upper -= 0x1a;
    /* Halfwidth and Fullwidth Forms, U+FF00 to U+FFEF */
    else if (code >= 0xff21 && code <= 0xff3a)
        *lower += 0x20;
    else if (code >= 0xff41 && code <= 0xff5a)
        *upper -= 0x20;
    /* Deseret, U+10400 to U+104FF */
    else if (code >= 0x10400 && code <= 0x10427)
        *lower += 0x28;
    else if (code >= 0x10428 && code <= 0x1044f)
        *upper -= 0x28;
}

void
XConvertCase(sym, lower, upper)
    register KeySym sym;
    KeySym *lower;
    KeySym *upper;
{
    /* Latin 1 keysym */
    if (sym < 0x100) {
        UCSConvertCase(sym, lower, upper);
return;
    }

    /* Unicode keysym */
    if ((sym & 0xff000000) == 0x01000000) {
        UCSConvertCase((sym & 0x00ffffff), lower, upper);
        *upper |= 0x01000000;
        *lower |= 0x01000000;
        return;
    }

    /* Legacy keysym */

    *lower = sym;
    *upper = sym;

    switch(sym >> 8) {
    case 1: /* Latin 2 */
/* Assume the KeySym is a legal value (ignore discontinuities) */
if (sym == XK_Aogonek)
    *lower = XK_aogonek;
else if (sym >= XK_Lstroke && sym <= XK_Sacute)
```

```
    *lower += (XK_lstroke - XK_Lstroke);
else if (sym >= XK_Scaron && sym <= XK_Zacute)
    *lower += (XK_scaron - XK_Scaron);
else if (sym >= XK_Zcaron && sym <= XK_Zabovedot)
    *lower += (XK_zcaron - XK_Zcaron);
else if (sym == XK_aogonek)
    *upper = XK_Aogonek;
else if (sym >= XK_lstroke && sym <= XK_sacute)
    *upper -= (XK_lstroke - XK_Lstroke);
else if (sym >= XK_scaron && sym <= XK_zacute)
    *upper -= (XK_scaron - XK_Scaron);
else if (sym >= XK_zcaron && sym <= XK_zabovedot)
    *upper -= (XK_zcaron - XK_Zcaron);
else if (sym >= XK_Racute && sym <= XK_Tcedilla)
    *lower += (XK_racute - XK_Racute);
else if (sym >= XK_racute && sym <= XK_tcedilla)
    *upper -= (XK_racute - XK_Racute);
break;
    case 2: /* Latin 3 */
/* Assume the KeySym is a legal value (ignore discontinuities) */
if (sym >= XK_Hstroke && sym <= XK_Hcircumflex)
    *lower += (XK_hstroke - XK_Hstroke);
else if (sym >= XK_Gbreve && sym <= XK_Jcircumflex)
    *lower += (XK_gbreve - XK_Gbreve);
else if (sym >= XK_hstroke && sym <= XK_hcircumflex)
    *upper -= (XK_hstroke - XK_Hstroke);
else if (sym >= XK_gbreve && sym <= XK_jcircumflex)
    *upper -= (XK_gbreve - XK_Gbreve);
else if (sym >= XK_Cabovedot && sym <= XK_Scircumflex)
    *lower += (XK_cabovedot - XK_Cabovedot);
else if (sym >= XK_cabovedot && sym <= XK_scircumflex)
    *upper -= (XK_cabovedot - XK_Cabovedot);
break;
    case 3: /* Latin 4 */
/* Assume the KeySym is a legal value (ignore discontinuities) */
if (sym >= XK_Rcedilla && sym <= XK_Tslash)
    *lower += (XK_rcedilla - XK_Rcedilla);
else if (sym >= XK_rcedilla && sym <= XK_tslash)
    *upper -= (XK_rcedilla - XK_Rcedilla);
else if (sym == XK_ENG)
    *lower = XK_eng;
else if (sym == XK_eng)
    *upper = XK_ENG;
else if (sym >= XK_Amacron && sym <= XK_Umacron)
    *lower += (XK_amacron - XK_Amacron);
else if (sym >= XK_amacron && sym <= XK_umacron)
    *upper -= (XK_amacron - XK_Amacron);
break;
    case 6: /* Cyrillic */
/* Assume the KeySym is a legal value (ignore discontinuities) */
if (sym >= XK_Serbian_DJE && sym <= XK_Serbian_DZE)
```

```
    *lower -= (XK_Serbian_DJE - XK_Serbian_dje);
else if (sym >= XK_Serbian_dje && sym <= XK_Serbian_dze)
    *upper += (XK_Serbian_DJE - XK_Serbian_dje);
else if (sym >= XK_Cyrillic_YU && sym <= XK_Cyrillic_HARDSIGN)
    *lower -= (XK_Cyrillic_YU - XK_Cyrillic_yu);
else if (sym >= XK_Cyrillic_yu && sym <= XK_Cyrillic_hardsign)
    *upper += (XK_Cyrillic_YU - XK_Cyrillic_yu);
        break;
    case 7: /* Greek */
/* Assume the KeySym is a legal value (ignore discontinuities) */
if (sym >= XK_Greek_ALPHAaccent && sym <= XK_Greek_OMEGAaccent)
    *lower += (XK_Greek_alphaaccent - XK_Greek_ALPHAaccent);
else if (sym >= XK_Greek_alphaaccent && sym <= XK_Greek_omegaaccent &&
 sym != XK_Greek_iotaaccentdieresis &&
 sym != XK_Greek_upsilonaccentdieresis)
    *upper -= (XK_Greek_alphaaccent - XK_Greek_ALPHAaccent);
else if (sym >= XK_Greek_ALPHA && sym <= XK_Greek_OMEGA)
    *lower += (XK_Greek_alpha - XK_Greek_ALPHA);
else if (sym >= XK_Greek_alpha && sym <= XK_Greek_omega &&
 sym != XK_Greek_finalsmallsigma)
    *upper -= (XK_Greek_alpha - XK_Greek_ALPHA);
        break;
    case 0x13: /* Latin 9 */
        if (sym == XK_OE)
            *lower = XK_oe;
        else if (sym == XK_oe)
            *upper = XK_OE;
        else if (sym == XK_Ydiaeresis)
            *lower = XK_ydiaeresis;
        break;
    }
}
```

Bibliography

[1] Lampson, B. W. (1973) "A Note on the Confinement Problem," *Commun. ACM*, **16**(10), pp. 613–615.
URL http://dx.doi.org/10.1145/362375.362389

[2] Getta, J. R. (2002) "Scrambling Covert Channels in Multilevel Secure Database Systems," in *SIS*, pp. 81–93.

[3] Kocher, P. C. (1996) "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, pp. 104–113.

[4] Wikipedia, "Keystroke logging," .
URL http://en.wikipedia.org/wiki/Keystroke_logging

[5] Ltd., K., "Key Ghost the Hardware Keylogger," .
URL http://www.keyghost.com

[6] Giani, A., V. H. Berk, and G. V. Cybenko (2006) "Data exfiltration and covert channels," in *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V. Edited by Carapezza, Edward M.. Proceedings of the SPIE, Volume 6201, pp. 620103 (2006).*, vol. 6201 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*.

[7] Shah, G., A. Molina, and M. Blaze (2006) "Keyboards and covert channels," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, USENIX Association, Berkeley, CA, USA, pp. p5–5.

[8] Bar-El, H. (2003), "Introduction to Side Channel Attacks - White Paper," .
URL http://www.discretix.com/PDF/Introduction%20to%20Side%20Channel%20Attacks.pdf

[9] Page, D. (2002), "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," Tech. report CSTR-02-003, Computer Science Dept., Univ. of Bristol, June 2002.

[10] Kocher, P. C., J. Jaffe, and B. Jun (1999) "Differential Power Analysis," in *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, Springer-Verlag, London, UK, pp. 388–397.

[11] Agrawal, D., B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM SideChannel(s):Attacks and Assessment. Methodologies," .
URL www.research.ibm.com/intsec/emf-paper.ps

[12] Percival, C. (2005), “Cache missing for fun and profit,” .
URL http://www.daemonology.net/papers/htt.pdf

[13] Brumley, D. and D. Boneh (2005) “Remote timing attacks are practical,” *Computer Networks*, **48**(5), pp. 701–716.

[14] Bernstein, D. J. (2004), “Cache-timing attacks on AES,” .
URL http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[15] Kelsey, J., B. Schneier, D. Wagner, and C. Hall (1998) “Side Channel Cryptanalysis of Product Ciphers,” in *ESORICS ’98: Proceedings of the 5th European Symposium on Research in Computer Security*, Springer-Verlag, London, UK, pp. 97–110.

[16] Trostle, J. T. (1998) “Timing Attacks Against Trusted Path,” in *IEEE Symposium on Security and Privacy*, pp. 125–134.

[17] Bagley, D., “X Windows System Lock Screen,” .
URL http://www.tux.org/b̃agleyd/xlockmore.html

[18] Tiri, K. (2007) “Side-Channel Attack Pitfalls,” in *Design Automation Conference*.

[19] Page, D. (2005), “Partitioned Cache Architecture as a Side Channel Defence Mechanism,” Cryptography ePrint Archive, Report 2005/280, August.

[20] Foundation, X., “X.Org Documentation,” .
URL http://wiki.x.org/wiki/Documentation

[21] Scheifler, R. W. (1998), “X Window System Protocol, X Consortium Standard, X Version 11, Release 6,” .
URL http://www.msu.edu/h̃untharo/xwin/docs/xwindows/PROTO.pdf

[22] Wikipedia, “X Window System core protocol,” .
URL http://en.wikipedia.org/wiki/X_Window_core_protocol

[23] Nye, A. (1992) *Xlib Programming Manual*, vol. 1, O’Reilly.

[24] Love, R. (2004) *Linux Kernel Development*, Developer’s Library.

[25] Intel (1997), “Using the RDTSC Instruction for Performance Monitoring,” .
URL http://cs.smu.ca/j̃amuir/rdtscpm1.pdf

[26] Vaughn, R., “Testing Times,” .
URL http://www.dodeca.co.uk/a_Testing_Times.htm

[27] “The Perl Homepage,” .
URL http://www.perl.org

[28] NIST, “The Expect Homepage,” .
URL http://expect.nist.gov

[29] Wikipedia, “Semaphore,” .
URL http://en.wikipedia.org/wiki/Semaphore_(programming)

[30] “POSIX Threads Programming,” .
URL https://computing.llnl.gov/tutorials/pthreads