# Timing Attacks on Secure Systems

Dan Halperin

April 13, 2006

## Contents

## List of Figures

## 1 Introduction

In class, we have examined a number of ways to break security systems. These methods mainly exploited cryptographic insecurities or abused protocols and mechanisms to subvert policy. For instance, DES was broken by high-performance brute forcing of the keyspace, WEP was broken by exploiting the allowed reuse of (cryptographically secure) RC4 keystream, and HDCP was broken by gathering enough data to allow us to span the module of keys. We also discussed the various insecurities in the mechanisms of key exchange and how, if these are also not performed very carefully, an eavesdropper can often obtain the shared secret.

In this paper, we analyze an entirely new class of attacks on systems previously thought to be cryptographically secure. These attacks use only the timing information of communications between parties to infer great amounts of detail about the information being transmitted, the identities of the participants, and can even recover the shared or private secrets of the entities involved. According to the literature, these attacks were pioneered in [4] in 1996 by Paul Kocher of Cryptography Research, Inc.

## 2 Plaintext Verification Attack

In this section, we present a very contrived example to show how external timing of secure mechanisms can leak information. Imagine we are attempting to authenticate to a server that we know stores passwords in plaintext. We submit a password guess to the server, the server compares this guess to the stored plaintext password, and sends us a response of either **true** if our guess matches the stored password or **false** otherwise. In our system we implement an authentication attempt as a system call on the local machine.

Assume that the hacker cannot acquire access to the server's password file. Additionally, assume that she cannot acquire other users' passwords by watching their authentication attempts, either because if she can snoop the attempts then they are signed in a cryptographically secure manner by the server's public key or, as in our example, are protected via operating system mechanisms that prevent inter-process information leakage. This system is secure against the types of attacks we have discussed thus far.

### 2.1 Description of the Attack

In this contrived example, we exploit the knowledge of how passwords are compared. The traditional algorithm for string comparison is described below.

**Algorithm 1** Compare strings $x$ and $y$

```
$i \Leftarrow 0$
while $x[i] = y[i]$ do
  if $x[i] = 0$ then
    return true
  else
    $i \Leftarrow i + 1$
  end if
end while
return false
```

The key to this attack is the observation that the string comparison operation runs in time linear in the length of the matching prefix of strings $x$ and $y$. Then suppose that the guess string $g$ contains the first $B$ characters of the password string $p$. To infer character $B + 1$, we measure the authentication time of all possible one character extensions of $g$, i.e. all strings $g'$ such that $g'[i] = g[i]$ for $i \in [0, B-1]$ and $g'[B] = c$ for any $c$ in the alphabet $\Sigma$. If we measure authentication time for all strings $g'$, the $g'$ with the correct guess of character $B + 1$ will run the longest. By starting with $B = 0$ and proceeding iteratively from there, we can infer the entire password using this technique.

### 2.2 Empirical Results

To verify that string comparison timing data does in fact allow us to recover the server's secret password string, we randomly generated passwords containing only lowercase characters of length varying between 5 and 14. We then ran the above hacking algorithm, both inferring the characters of the password one by one as well as determining successfully when the end of the password was reached. The source code for this program can be seen in the Appendix A.1.

However, there are many sources of variance in the timing data. Compiler optimizations, cache hits and misses, out-of-order execution, pipelining, branch prediction, context switching, and other unknowns can cause drastic differences in the timing of any given authentication attempt. We therefore employed a number of ad-hoc tricks in the code for this program (see A.1) to enhance its accuracy, such as priming the hardware by attempting to authenticate with a password once before timing the authentication attempt or running each password guess multiple times and taking the minimum time over a large number of runs. Some of these tricks were taken from the literature [3] and some were of original design.

Figure 1: A successful and a failed attempt to infer passwords

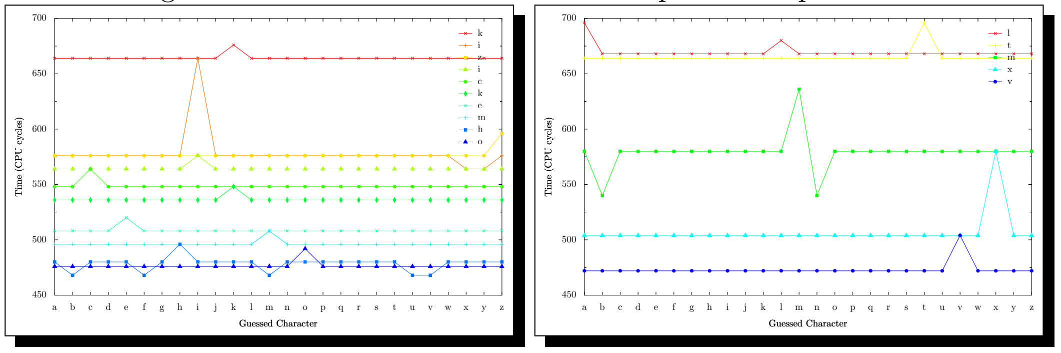


Figure 1 shows the timing measurements made by the program during a successful attempt to hack the password `ohmekciziks` and an unsuccessful attempt to hack the password `vxmtlhhvt`. The graph shows that the real-time performance of the string comparison algorithm closely approximates the theory described above. In general, the comparison times are monotonically increasing as the length of the correct prefix gets longer, and it is also usually true that the comparison time for a correct guess of length $B$ is equal to the comparison time for an incorrect guess of length $B + 1$. These both parallel the theoretical intuition developed above.

However, there are also some interesting anomalies observed in the figure. Note that the authentication times for guesses of length $B + 1$ are occasionally equal to or even sometimes less than the times for the incorrect guesses of length $B$, such as when inferring the first character `o` in the successful attempt. Sometimes the computation time for a guess of length $B + 1$ is not equal to the computation time of the correct attempt of length $B$ but is instead even greater. As we see in the final, incorrect inference made in the failed attempt, for some reason the computation spiked when guessing `a`, which overshadowed the expected increase in runtime when it was guessed correctly that the next character `l`. Intuitively, we believe that these anomalies are most likely due to branch prediction, pipelining, loop unrolling, and perhaps out-of-order execution on the processor.

Finally, note that the the comparison time increases significantly more between password guesses of length 8 and length 9 than between shorter passwords. Given the away that C stores arrays in memory, the first 8 contiguous characters occupy 64 bytes of memory. This dramatic increase in time could be explained if the 8th and 9th characters of the password fell on different cache lines, and upon inspection the cache line size on the test machine is indeed 64 bytes. Thus it is likely, as Kocher suggested in [4], that implementation-specific timing information can even yield insights into the hardware configuration of the authenticating machine.

## 2.3 Defense Against the Attack

This particular timing attack is very easy to defend against. Currently, the authentication process simply takes the guess $g$ and the password $p$ and compares $g$ to $p$. Consider instead implementing the authentication by using some hash function $H$ to transform the strings and comparing $H(g)$ to

$H(p)$. Any hash function $H$ where a common prefix in $g$ and $p$ does not lead to a common prefix of $H(g)$ and $H(p)$ will suffice to defeat our attack.

In the case described above, $H$ is the identity function. Another common hash function, used in older versions of the AOL Instant Messenger client, is to XOR the string with a fixed constant. Neither of these hash functions satisfies the above criterion. However, there is an abundance of hash functions that do satisfy this condition, with two of the most commonly used ones being MD5 and SHA-1. However, as we show in the next section, even cryptographically secure hash functions such as RSA can be broken with cleverly designed timing attacks against undefended implementations.

## 3 RSA Attack

The RSA hash function is one of many hash functions that defeats the trivial timing attack presented in the previous section. In [2], David Brumley and Dan Boneh of Stanford University showed how to recover the private key used in the default implementation (at the time) of OpenSSL in Linux operating systems. They were able to extract the SSL private key not only between processes on a singly machine as in our contrived attack, but also across a Stanford campus-wide network on a web server (Apache with mod_SSL) and an SSL-tunnel.

### 3.1 The RSA Algorithm

This presentation of the RSA algorithm is taken from [1].

**Algorithm 2** The RSA Exponentiation Cipher

Choose $p$ and $q$ two large primes, and let $N = pq$. Define $\phi$ as the Euler totient function where $\phi(N)$ is the number of numbers less than $N$ that are relatively prime to $N$. It is a simple result from number theory that $\phi(n) = (p-1)(q-1)$.
To support encryption and decryption, choose an integer $e < N$ that is relatively prime to $\phi(N)$, and find a second integer $d$ such that $ed \bmod \phi(N) = 1$. The public key is $(e, N)$ and the private key is $d$.
If $m < N$ is the plaintext of a message, then the ciphertext $c$ is

$$c = m^e \bmod N$$

and to decrypt a given ciphertext we use the result that

$$c^d \bmod N = m.$$

If we wish to encrypt a message that is longer than $\log N$, we simply encrypt it in blocks consisting of $\log N$ bits each. Decryption is similar.

The majority of the work in the RSA algorithm is precomputed when the public and private keys are generated. In order to actually encrypt or decrypt secret messages, all that is done is the modular exponentiation.

### 3.2 Description of the Attack

Public-key cryptography is orders of magnitude slower than symmetric key cryptography [6]. This holds true for RSA as well – modular exponentiation is a costly operation – and consequently several number-theoretic optimizations are employed to speed computation time.

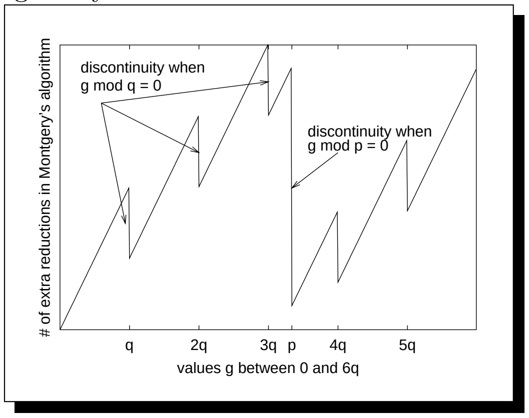
The most basic optimization uses the Chinese Remainder Theorem to obtain a factor of 4 speedup in computation time. In essence, this optimization calculates

$$m_1 = c^{d_1} \bmod p \quad \text{and} \quad m_2 = c^{d_2} \bmod q$$

and combines $m_1$ and $m_2$ to find $m$. Here, $d_1$ and $d_2$ are precomputed from $d$, $p$, and $q$. The key to the attack lies in recovering one of $p$ or $q$ and then exploiting the fact that this allows us to recover $\phi(N)$ and then to infer the private key $d$. This unlocks all encrypted communications to the server and facilitates spoofing by allowing a malicious entity to generate and cryptographically sign arbitrary messages.

One optimization used in speeding up the modular exponentiation itself is called a Montgomery reduction. Werner Schindler discovered that there is a relationship between the number of Montgomery reductions employed in calculating $g^d \bmod h$ and how close $g$ is to a multiple of $h$, as shown in Figure 2, copied from [2]. The key insight here is that the sharp drop in the number of Montgomery reductions as $g$ switches from being slightly less than a multiple of $h$ to slightly more than a multiple of $h$ is observable in the timing information. In the RSA algorithm, this allows us to tell how close $g$ is to a multiple of one of the factors $p$ or $q$ of $N$.

Figure 2: Number of Montgomery reductions as a function of the input $g$ and the modulus $q$



One final optimization that the authors exploited was to notice that OpenSSL used Karatsuba's $O(n^{\log_2 3}) \approx O(n^1.58)$ algorithm to multiply two integers of the same length in number of bits and the standard $O(nm)$ algorithm to multiply two integers of differing sizes $n$ and $m$. By combining their implementation-specific knowledge of the workings of OpenSSL 0.9.7 and the timing characteristics resulting from the implemented optimizations, Brumley and Boneh were able to successfully iteratively infer the bits of the smaller prime factor of $N$ in a manner similar to that described in the previous section. The ultimate result was that across multiple links in Stanford's campus network, they were able to recover the 1024-bit private key of an OpenSSL 0.9.7. server in about 2 hours. Their paper [2] also describes several other pieces of information that can be inferred such as the underlying architecture of the system.

## 3.3 Defense Against the Attack

Brumley and Boneh present three defenses against this attack, but the simplest and extremely effective technique is called RSA Blinding. If $g = m^e \bmod N$ is the ciphertext to be decrypted by the server, the RSA blinding operation will instead calculate

$$x = r^e g = r^e m^e = (rm)^e \bmod N$$

where $x$ is random, and then after decrypting $x$, divide the result by $r$ to get the plaintext $m$. The attack described above is a chosen ciphertext attack and is thus completely defeated by this RSA blinding technique.

## 4 Inter-Keystroke Timing Analysis of SSH

In this section we discuss an entirely new way to use timing data to compromise secure systems. In [7], Dawn Song, David Wagner, and Xuqing Tian at the University of California, Berkeley present ways to exploit two weaknesses in SSH that lead to serious security risks. The first weakness is that (when using a block cipher) the transmitted packets are padded to 8-byte boundaries, revealing the approximate size of the transmission and leaking information such as whether the length of the authenticating user's password is at least 8 characters. This also allows for easier development of traffic signatures for specific types of behavior. The second, and more interesting weakness in the SSH protocol is that, in interactive mode, every individual keystroke generates a separate IP packet sent to the remote machine. The authors of this paper develop some very interesting attacks using this property.

### 4.1 Keystroke Packet Analysis

Consider the possible meanings of a keystroke when logged into a server via SSH. The keystroke could be a character typed on the command line or in a text editor, which is displayed on the screen immediately upon being type. It could also be a character of a password, which will not be echoed by the server. A third possibility, among many more, is that it is a command in a mail client, and will cause redrawing of many parts of the screen.

A packet containing a single keystroke has size 20 bytes. If there is a response from the server of exactly 20 bytes, then it can be presumed that the purpose of that keystroke was to submit that character as visual text. If there is no response from the server, then that key press was masked and may well have been part of an entered password or passphrase. And finally, if there was a large response from the server that key press could have stood for a command in a larger context of a running application.

Figure 3: Traffic signature of su in an SSH session.

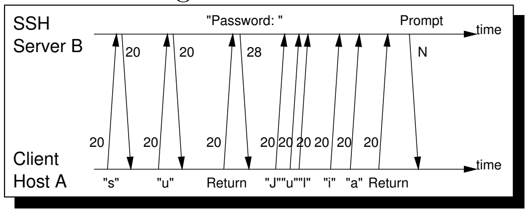


Figure 3 copied from [7] shows the traffic signature associated with running `su` in an SSH session. Given this information, an intelligent eavesdropper might be able to guess with some confidence that the root password, `Julia`, is five characters long. This confidence level can be significantly increased if the adversary has an account on the system that can run `ps` and see what process the user is running. A third situation vulnerable to this type of analysis is a nested SSH connection, where the user first authenticates to one SSH server and then initiates a new SSH connection from that server to a third machine. Because of the details of the SSH protocol, the keystrokes from the user's computer to the first server are sent one-by-one interactively whereas the keystrokes are sent from the first server to the second in one single packet. Nested SSH connections are easily recognized [7] and this analysis yields the length of the user's password on the second machine.

However, the most significant security risk exposed in this paper is that in interactive mode the inter-packet timings extremely closely approximate the inter-keystroke timings – packet generation and emission time is insignificant compared to the physical delay in human typing speeds.

The authors of the referenced paper were able to exploit this information to obtain a dramatic improvement in password-cracking time.

## 4.2 Description of the Attack

As Song et al. reference in this paper, multiple studies have been released that use keystroke timing for biometrics: the duration of key strokes and latencies between key presses can be used as identifying characteristics of a keyboard user. In this paper, they turn that research on its head and show that the inter-keystroke timing information, which is equal to the previously mentioned latency between key presses, can be used to infer information about the characters being typed themselves.

The way this works is that passwords are peculiar in that they are typed so frequently that the pattern of hand and finger movements on the password is acquired by physical memory and that these movements are constrained only by the physical limits on the user's typing speed. The authors took multiple experienced keyboard users (using the same keyboard layout) and measured how fast their fingers could physically type certain key combinations. What they found is that these times reveal easily identified patterns about what characters are being typed. For instance, two characters typed by different hands will be typed the fastest, two keys typed by different fingers on the same hand take slightly more time to type, and that by far the slowest two-character combinations are those where one single finger must type two different keys. Song et al. then performed a comparative study and showed that timing information from one typist can be applied to analyze the timings of a different user as long as both users have some minimal level of typing proficiency and use the same keyboard layout.

The problem, then, is to guess what the password is given only the inter-keystroke timings. The authors applied an Artificial Intelligence technique called a Hidden Markov Model which will probabilistically rank the characters being typed using the fact that consecutive times are related – two times come from three key presses, where the same key is the second key pressed in the first combination and the first key pressed in the second. As they also know the length of the user's password, the use this Hidden Markov Model to probabilistically rank all passwords of that length, greatly speeding up the time needed to brute-force the password of the target user. On their tests, using passwords of length 8, in the average case the user's actual password fell in the top 2% of the probabilistically ranked list of all 8-character passwords. This yields a factor of 50 speed-up in the time needed to crack the user's password in the average case.

## 4.3 Defense Against the Attack

As in the previously described timing attacks, there are simple countermeasures that can be implemented against this analysis. The first is to send dummy packets back when character echo is turned off – this minimizes the traffic signature of commands that require passwords such as `su` or `pgp`. The second is to have a metric for when the connection is active, say, whenever the inter-packet delay is less than 500 milliseconds, and send traffic at a constant rate when the link is active. If this rate high enough, there will not be any noticeable delay in the user's typing, but the timings of the outgoing packets will no longer reflect the physical speed at which keys are typed. These defenses fully defeat the attacks discussed in this section.

## 5 Conclusions

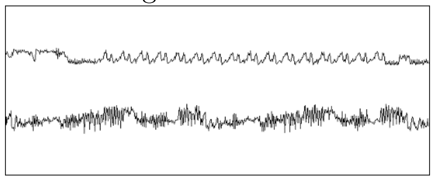
In this paper we have presented a new class of attacks on secure systems. The take-home message here is that it confirms what we already know: **Security is *Hard*.** The attacked systems are not cryptographically insecure – this paper did not show that RSA is broken – but rather that the cryptographically secure mechanisms and protocols underlying these systems were implemented in such a way as to unknowingly reveal private information. The possibility of timing attacks adds another layer of complexity to the development of a secure system.

The information revealed by these types of attacks is extremely variable. Among the information revealed is the obvious – the password of an authenticating user or the private key used for public key encryption, but it was also shown that this information can yield insights into the system architecture on which the software is run as well as the identity of the user running an SSH client. It is conceivable that there is more information embedded in this data which we do not yet know how to find.

The common thread among the attacks presented in this paper is that they are practical attacks that work on actual systems. The origin of timing attacks was theoretical; no one actually believed that they could be performed in real live systems, and before the paper of Brumley and Bohen the RSA blinding technique, which had been known several years beforehand, was not turned on by default in many implementations of SSL. Even though the potential for a vulnerability was known, the performance hit taken when securing the system was deemed too great without confirmed evidence of a real attack.

For future work, it should be noted that timing attacks are not the only types of attacks of this class. Paul Kocher, the originator of these types of analyses of secure systems, published another paper [5] in which he introduces differential power analysis (DPA). DPA takes cryptographic hardware, such as smart cards, and analyzes the power drain of the system to discover both what cryptographic algorithms are being used and which computation paths are being taken in using them. The below Figure 4 from that paper shows power analysis on a smart card. The first row shows the entire power trace in which all 16 rounds of DES are clearly visible, and the second row is a zoomed-in view of the second and third rounds. Then using this data, known ciphertext, and the known computation paths inferred from this data, the symmetric key, one copy of which is stored on the card, can be learned. The implications of this work are that even after a secure system is developed (and a *log* of work was put into the math behind the DES algorithm), there are still a number of ways in which the algorithm can be implemented insecurely and there may be new unexpected ways to attack these secure systems that have yet to be discovered. The problem of security may be a race with no end, but really interesting research and really cool results can be found along the way.

Figure 4: Power usage of a smart card running DES



The figure above shows SPA monitoring from a single DES operation performed by a typical smart card. The upper trace shows the entire encryption operation, including the initial permutation, the 16 DES rounds, and the final permutation. The lower trace is a detailed view of the second and third rounds.

## References

[1] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[2] D. Boneh and D. Brumley. Remote timing attacks are practical. *Proceedings of the 12th USENIX Security Symposium*, 2003.

[3] Intel. Using the rdtsc instruction for performance monitoring. *Intel*, 1997.

[4] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Proceedings of Cryptography*, CRYPTO 96(LNCS 1109):104–113, 1996.

[5] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[6] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* John Wiley & Sons, Inc., 1996.

[7] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. *Tenth USENIX Security Symposium*, 2001.

## A Appendix

### A.1 String Comparison Hacking Code

```
/**
 * Dan Halperin
 * password.c
 *
 * This file contains a quick-and-dirty implementation of a timing-based
 * password cracker when strcmp() is used as the password verification
 * function. The basic philosophy is that strcmp will run in time linear in
 * the number of correct characters at the start of a password guess. Thus,
 * if we know the first B bytes of the password, to infer byte B+1 we need
 * only guess all (in this case 26) possible succeeding bytes and measure
 * which guess takes the longest to be compared with the password.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

// A global variable that holds the password in plaintext.
char password[16];
unsigned int guess_count;

// The function that checks the password.
bool check_password(char* guess);
// The function that attempts to infer the password as described above.
bool hack_the_password();

int main()
{
    // Set the seed for the random number generator
    srandom(time(NULL));

    // Loop  forevever, generating and hacking passwords
    while (true)
    {
        // Choose a random password length in [5, 13]
        int password_length = random() % 9 + 5;

        // Assign random characters in 'a'-'z' to the password
        for (int i = 0; i < password_length; ++i)
            password[i] = (char) (random() % 26 + 'a');
        // Null-terminate the password
        password[password_length] = '\0';
        // Zero guesses so far.
        guess_count = 0;

        // Print the password
        printf("%15s: ", password); fflush(stdout);

        // Run a _very_ crude attempt to hack the password, that does no
        // error correction nor probabilistic analysis and count the number
        // of tries we need to correct infer all password_length characters
        // of the password.
        int count = 0;
        while (!hack_the_password())
        {
            if (count % 10 == 0 && count > 0)
                printf("."); fflush(stdout);
            ++count;
        }
        printf("\tFail count: %d\t Guesses: %d.\n", count, guess_count);
    }

    return 0;
}

// This function is merely a wrapper for strcmp. This functionality could
// be extended later to support other methods of authentication.
bool check_password(char* guess)
{
    return (strcmp(password,guess) == 0);
}

// A wrapper for the cpu instruction rdtsc, which stores the current cycle
// number as a 64 bit number. The cpuid instruction is included because it
// forces inorder execution. (See the 1997 Intel document on using rdtsc
// for performance testing cited in the bibliography).
static inline unsigned long long read_rdtsc(void)
{
    unsigned long long d;
    __asm__ __volatile__ ("cpuid \n rdtsc" : "=A" (d) );
    return d;
}

// Attempts to hack the password.
bool hack_the_password()
{
    // Array containing the current guess of the password
    char current_guess[16];
    current_guess[0] = '\0';

    // max_char stores the character which is most likely to be byte B+1 of
    // .. the password, as the guess ending in max_char takes the longest
    // .. time to compute.
    char max_char;
    unsigned long long max_time;

    // min_time is the minimum time it takes to authenticate a given
    // .. password. This will help us to get around the problems of context
    // .. switching increasing computation time.
    unsigned long long min_time;
    unsigned long long start, end;

    // Infer the (up to) 15 characters of the password one by one.
    for (int cur_char = 0; cur_char < 15; ++cur_char)
    {
        // Null-terminate the string
        current_guess[cur_char + 1] = '\0';

        // Initialize the maximum time to 0 and set max_char='a' by default
        max_char = 'a';
        max_time = 0;

        // Time each of the characters
        for (char guess = 'a'; guess <= 'z'; ++guess)
        {
            // Make the guess
            current_guess[cur_char] = guess;
            // Initialize min to big!
            min_time = (1 << 30);

            // Repeat 20 times: first, prime the cpu by guessing the
            // password 2 times, then actually time the compare process
            for (int i = 0; i < 20; ++i)
            {
                ++guess_count;
                // If we have found the correct password, we're done
                if(check_password(current_guess))
                    return true;
                ++guess_count;
                check_password(current_guess);
                ++guess_count;
                check_password(current_guess);

                // Time the password guess
                guess_count += 3;
                start = read_rdtsc();
                check_password(current_guess);
                check_password(current_guess);
                check_password(current_guess);
```

```
                end = read_rdtsc();

                // If this was the shortest time for this password that
                // we've seen, awesome!
                if (end - start < min_time)
                    min_time = end - start;
            }
            if (min_time > max_time)
            {
                max_char = guess;
                max_time = min_time;
            }
        }
        current_guess[cur_char] = max_char;

        if (max_char != password[cur_char])
            return false;
    }
    return false;
}
```

## A.2 A Hacking Run (low load)

```
     rrrrvliuzb:        Fail count: 0    Guesses: 28201.
         qaqphj:        Fail count: 4    Guesses: 79081.
      qrtxzvntc:        Fail count: 2    Guesses: 59521.
         mqudho:        Fail count: 1    Guesses: 26641.
      cbgychgsh:        Fail count: 0    Guesses: 25801.
   hoflgzefyztt:        Fail count: 1    Guesses: 52201.
       zguvlbjk:        Fail count: 0    Guesses: 23041.
  mqgoynjfjswrd:        Fail count: 0    Guesses: 37801.
      qjiqjdjmc:        Fail count: 0    Guesses: 25201.
   ixdkjnsvfymd:        Fail count: 0    Guesses: 34681.
        vlxnhot:        Fail count: 0    Guesses: 21001.
   gctynwhcybkw:        Fail count: 0    Guesses: 36961.
     whtoeynsea:        Fail count: 1    Guesses: 49921.
       pzdwpyhv:        Fail count: 1    Guesses: 46201.
       bvqxfswg:        Fail count: 0    Guesses: 22561.
    umbdhphheal:        Fail count: 0    Guesses: 32521.
  pcguywsitvjpl:        Fail count: 7    Guesses: 179161.
         ufgaja:        Fail count: 0    Guesses: 15601.
          kfvcp:        Fail count: 0    Guesses: 14281.
         grongs:        Fail count: 6    Guesses: 108241.
```

## A.3 A Hacking Run (high load)

```
         chtrhs:        Fail count: 0    Guesses: 17761.
  djvbdpmsozvst: .....  Fail count: 51   Guesses: 1116121.
  qqnwmdylmppvk: .....  Fail count: 51   Guesses: 1289761.
          ccdhl:        Fail count: 0    Guesses: 13801.
         lrpxle:        Fail count: 0    Guesses: 16081.
          gwuko:        Fail count: 4    Guesses: 51601.
       xkzdkkpb: .      Fail count: 11   Guesses: 124921.
     mlhqpmybnl:        Fail count: 0    Guesses: 29401.
      ckfijofdy:        Fail count: 0    Guesses: 27841.
   pwiobsbsubeh:        Fail count: 0    Guesses: 35161.
     wxyualhuqs: .      Fail count: 11   Guesses: 304801.
  ybpdgqbxojopd:        Fail count: 0    Guesses: 37801.
    hxslhcheceg:        Fail count: 0    Guesses: 31921.
      ncgfdeivk: .      Fail count: 11   Guesses: 300721.
     llozucoztx:        Fail count: 0    Guesses: 30841.
        lieprkr:        Fail count: 0    Guesses: 20761.
          qhlsn:        Fail count: 0    Guesses: 14041.
       xsbscrfn:        Fail count: 0    Guesses: 23401.
          gjjwi:        Fail count: 0    Guesses: 13441.
          thndn:        Fail count: 0    Guesses: 14041.
```