

# Experiments in Remote Timing Attacks

Evrhet Milam    Joe Russell    Aaron Fisher

## Abstract

Timing attacks are a side-channel attack that avoid trying to break cryptographic algorithms and instead breaks the implementation of the algorithm. By looking at how long an algorithm is able to execute on a set of data we are able to determine a password with a very limited number of tries. Our experiments show that we can easily crack weak password implementations and timing attacks can be used on stronger algorithms such as HMACs to gain access to private data. This is possible over the Internet and not limited to having local access to the computer under attack.

# 1 Introduction

Attackers have discovered that, because most of today's modern encryption algorithms are so strong – even considered to be unbreakable, many times the actual encryption device is easier to exploit than trying to break the encryption algorithm through cryptanalysis. In this sense, the encryption device is defined as the device that receives the plaintext as input, processes the input with some form of encryption, and outputs cyphertext (or processes cyphertext in the reverse order).

When the encryption device (such as a web server) receives input, like a user's password, hashes the password, compares the hash to a stored value, and then returns the result, the device naturally produces “side channel information.” Side channel information can be any type information that the attacker uses to gain insight on the internal design of the device. Some examples are: timing information pertaining to the amount of time operations take, power consumption, and even sound emission. By manipulating inputs and monitoring side channel information, attackers can map the processes going on within the device and try to navigate through the encryption process or, theoretically, discover the entire encryption algorithm.

These attacks are known as “Side Channel Attacks.” There are many different types of side channel attacks, and probably more types will emerge in the future, but some of the better-recognized attacks are the power consumption attack and the timing attack. We focused our research on the implementation of a timing attack, but we will present the basic concepts of different types of side channel attacks as well.

## **Power Consumption Attacks**

The power consumption attack comes in two varieties. The simple power analysis and the differential power analysis. In the simple power analysis attack, the attacker analyzes the power consumption of the system while it performs the encryption process. The microprocessor in the system will use different amounts of power to perform different instructions. Through this analysis the attacker can see the type of operation and the sequence of operations, thereby discovering the encryption key. According to Paul Kocher, a cryptanalyst considered to be a pioneer of side channel attacks, most systems are vulnerable to this type of attack; however it requires access to the device’s power or ground input and can be easily guarded against by intentionally adding voltage noise to the system (Kocher, 2009)<sup>i</sup>.

The differential power analysis is theoretically the same, only it can compensate for error caused by additional noise through statistical analysis. To provide accurate statistical analysis, this approach requires a larger sample size of data.

## **Timing Attacks**

Timing attacks involve looking at the amount of time it takes to execute a cryptographic algorithm or other authentication algorithm. This often gives information on the type of CPU, the algorithm in use, the operating system, or other implementation details.

Knowing the algorithm and implementation that is in use, we can use a timing attack to break into an existing system or gain access to a private key. This has happened with systems such as Google's keyczar<sup>ii</sup>(N. Lawson, 2009), OpenID, any Message Digest code in Java(C. Hale, 2009)<sup>iii</sup>, and OpenSSL implementations used on web servers such as Apache (D. Boneh, 2003)<sup>iv</sup>.

Many online systems give each user their own identity and the user is authenticated with a username and password or another unique form of credentials. One of the most common authentication methods is a HMAC

(hashed message authentication code). This is a message authentication code that has been hashed using a hashing algorithm such as MD5 or SHA-1.

To make sure a message is sent from the actual sender, the sender attaches the message along with the message hashed with a secret key producing a HMAC. This secret key is usually only known by the sender and the recipient. The recipient is then able to calculate the same HMAC using the message and the secret key. If the HMACs match the recipient knows the message was sent from the real user.

## **Timing Attacks in Detail**

The timing attack happens during this comparison period between the supplied HMAC and the computed HMAC. SHA-1 and MD5 HMACs will take the same amount of time to compute from the same length message. The difference in the timing comes in comparing one HMAC to a different HMAC because of optimizations in byte and string comparison functions. This not only holds true for comparing HMACs but comparing any two strings such as a password to another password. When comparing these HMAC signatures the operation is simply to do a string comparison one character at a time. This can be outlined in the following code:

```
if (user_password == supplied_password) {
    performAction();
} else {
    accessDenied();
}
```

This simple ‘==’ comparison is broken down into checking each character by looping over the characters and making sure they are the same. This algorithm is usually the same over most programming languages. The following code outlines how most programming languages do a simple string comparison.

```
boolean compareStrings(string lhs, string rhs) {
    for (int x = 0; x < lhs.length; x++) {
        if (lhs[x] != rhs[x])
            return false;
    }
    return true;
}
```

Many programming languages do some optimizations above this, but they usually have a similar “short circuit” point if at some time the strings don’t match. It does not make sense to keep checking if the strings are equal if you already know they are not the same.

To continue with our example if two strings were compared using this algorithm:

```
user_password = 'password'
supplied_password = 'pass1234'
```

The `compareStrings` function would return false, and the `accessDenied` function would be executed. The function would return false after comparing only five characters ending on comparing the 'w' and the '1'. The 'rd' and '24' would never be compared. Again to reiterate, we would stop after we knew the strings were different which would be before we finished comparing the whole string.

The attack comes into play by looking at how long it takes to get denied from the server. Even though the loop and comparing strings are very efficient there is still micro second difference in comparing five letters to comparing six letters. This is what leads to the timing attack.

## **A simple timing attack**

Continuing from our example in the previous section, if Alice usually gives a secret password of a fixed length to Bob to login to his server, and he uses a simple string comparison to compare the provided secret to the stored secret password then Mallory can use a timing attack to get the password.

Mallory starts by checking each possible character as the first character of Alice's password (the rest of the characters do not matter until we know what the first character is). After finding the first character, Mallory can move on to the next character and find that letter, and iterate to the last character until she

has figured out the whole password. For example if Mallory knows the password characters fall between a-z, she starts by testing each letter a-z.

```
axxxxxxx  
bxxxxxxx  
cxxxxxxx  
...  
yxxxxxxx  
zxxxxxxx
```

By measuring the time of each attempt Mallory can see which one took the longest amount of time, and this is the correct character. The reason Bob's server took longer is because two characters were tested. For example if the password was 'password' and Mallory supplied 'pxxxxxxx', Bob's server would have compared the 'p' and 'x', and then compared 'a' and 'x'. This extra comparison takes a noticeable amount of time. Mallory can then move on to testing the second character of the password with the first character as 'p', and use the same method for each character.

One issue that comes up is that there is no way Bob's server or the connection between Bob and Mallory is perfectly stable. There is always going to be some differences in how long it takes packets to travel from Bob to Mallory. To make up for this jitter in the connection, Mallory can try the same password multiple times. Depending on the connection this could be between 100 to 1000 times, and is talked extensively by Crosby(Crosby, 2009)<sup>v</sup>. After the multiple

tries, Mallory can use statistics to minimize the jitter and figure out which character is right.

This seems similar to a brute force attack, but we are able to guess the password with fewer attempts because we are given a hint of what the character is, and do not need to test every possible combination. Instead of testing on average  $8^{24}/2$  combinations, we can test  $8*24*1000$  combinations, which is a significantly lower number. With HMACs this becomes a bigger deal because HMACs have a wider character range and are significantly longer than eight bytes. MD5 and SHA-1 HMACs are usually 16 bytes to 20 bytes respectively.

The timing attacks we focused on were remote timing attacks over the Internet. Timing attacks have been studied for a long time, but most of the resources we gathered were from Crosby's paper "Opportunities and Limits of Remote Timing Attacks". Crosby goes into detail on how to deal with jitter, and best filter the jitter out of the attack using "percentile filters"

## **2 Discussion**

Armed with the knowledge that timing attacks have been successfully executed on the Internet, we decided to implement our own timing attack algorithm. Our goal was to write our own code to see if we could implement a

timing attack in a simple login environment on our own machine. In this controlled environment, we could better study timing attacks. The code is found in Appendix A and Appendix B. The login server was a socket server that would take a string input, check to see if the input string matched the string we had hard coded into it, and would short circuit as soon as it found a letter that was wrong by returning the string “Incorrect Password”, or “Correct Password” if the two strings matched.

Our first algorithm took each letter of the alphabet, built a string for each letter and iteration, and would attempt each string 1000 times and average the response time for each result. “aaaaaaa” went first and was timed 1000 times, then “baaaaaa” went, etc. The average response times were then compared and the response time with the slowest average response time “won”, and the second letter was measured in the same fashion, using the assumed correct first letter to win as the first letter in the new string.

We were unable to get the first algorithm to successfully execute a timing attack. So after much analysis and tweaking, we realized that doing each test in order (ie. “A” 1000 times, *then* “B” 1000 times, etc...) could ruin the results because the server could be faster or slower at different times. We changed the algorithm to alternate through letters (ie. “A” one time, then “B” one time, all the

way to the end of the alphabet, then repeat). This was a theoretical improvement that we kept in our final implementation, but the second iteration was unable to successfully execute a timing attack.

The third and final implementation required us to go back to academic articles to research timing attacks. We found the solution to our problem in “Opportunities and Limits of Remote Timing Attacks” (Crosby, 2009). Instead of taking the average of all of the response times for each letter, we learned that we should do a percentile filter to remove jitter. We did a percentile filter by taking the response time from each letter located at 5% from the longest time. For example, if there were 1000 tests for each letter, we would take the 50th slowest response time from each letter and compare them to each other and take the slowest time as the correct letter. This percentile filter inherently throws out outliers and is an accurate way to analyze response times. After implementing the filter, we were able to successfully execute a timing attack on our local server, and on a server over a local area network.

### **3 Results**

In the graphs below is an example of the response times of a correct letter (figure 3.1) and an incorrect letter(figure 3.2) over 1000 attempts. To compare

the two, the algorithm performs a percentile filter by grabbing the number at the 95<sup>th</sup> percentile for each letter (in this case, the response time of the 5<sup>th</sup> slowest percentile attempt, or 95 on the graph) and sort those response times. After these response times have been sorted, the algorithm chooses the slowest response time, and we know it should be the correct character. In this case, X has the slowest response, and is chosen. If we were to include the other graphs, they would look similar to the graph of W, since the incorrect letters have about the same response time, while the correct letter should have a slightly higher response time.

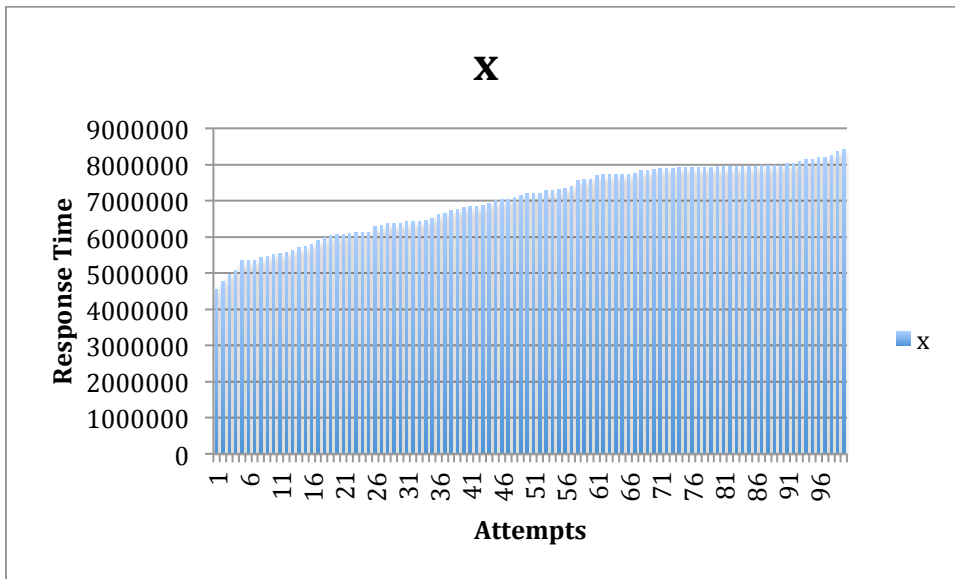


Figure 3.1 Example of correct letter response time

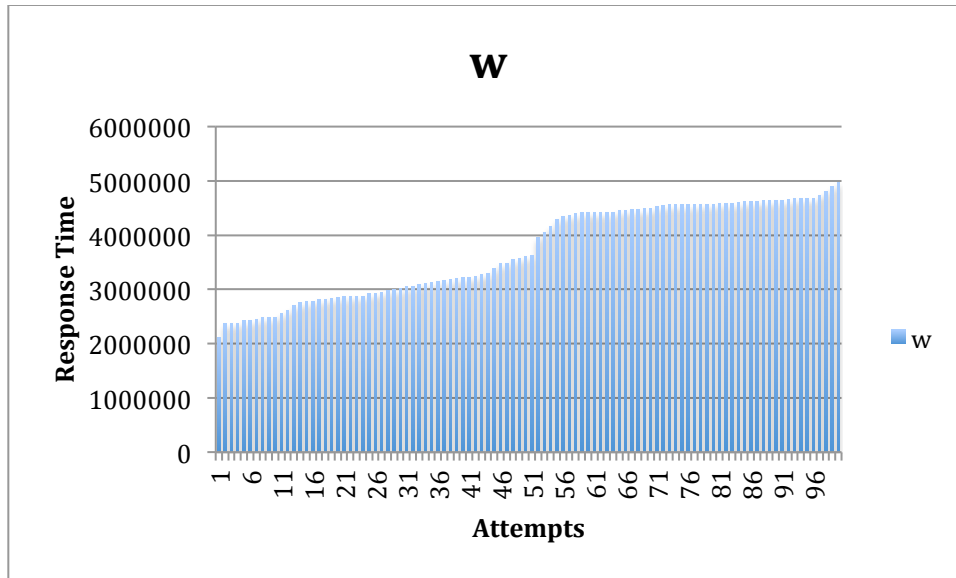


Figure 3.2 Example of incorrect letter response time

We were able to write an algorithm that successfully executes a timing attack on a server with a password of eight letters. In the future, we would like to expand our tests to the Internet; academics and hackers have already successfully executed timing attacks over the Internet. Timing attacks are not theoretical, and can be executed on vulnerable servers.

## 4 Recommendations

The timing attacks are an issue because an attacker is able to detect the difference in how long each string combination takes to compare against the real key. This is because of the optimization in string comparisons, and short-circuiting as soon as we know the strings are not equal. To fix this, short-circuits

should be removed from the string comparison. The following code gives an example of string comparison that does not have the same timing-attack issue.

```
boolean compareStrings(string lhs, string rhs) {
    if (lhs.length() != rhs.length())
        return false;
    boolean equal = true;
    for (int x = 0; x < lhs.length; x++) {
        if (lhs[x] != rhs[x])
            equal = false;
    }
    return equal;
}
```

This string function will compare each character even if it knows the strings are not equal. This makes the function take the same amount of time no matter what string is provided.

Timing attacks are very subtle and can be found in other parts of the implementation of an algorithm. Most code that is already part of the programming language or of an existing library is going to be optimized to take as little time as possible. To fully remove all issues of timing attacks an expert should be brought in to work through the code and look through all existing issues.

If possible it is best to use an already implemented library for cryptography that has had timing attacks removed. This is very challenging code

to audit, and it should be left to professionals instead of attempting to implement the code on each project.

Hagai Bar-El, in his white paper “Introduction to Side Channel Attacks”, recommends some similar tactics (H. Bar-El)<sup>vi</sup>.

- Data Independent Calculations
- Avoid conditional statement optimizations
- Adding Delays / Time Equalization
- Power Consumption Balancing
- Add “Noise”

All of these focus on making algorithms take a random or constant amount of time. The goal is to make sure attackers cannot glean any information by analyzing how long a response takes.

## 5 Conclusion

We have shown that remote timing attacks are a practical attack against improperly implemented algorithms in existing servers. Our limited eight-character attack can be used against even longer passwords or keys. Over time more and more exploits will be written to take advantage of timing attacks. Languages such as Java, Ruby, Python, and all cryptography libraries need to take the time to secure their code against timing attacks. There are many

optimizations and short circuits, which make timing attacks possible. It is better if this is fixed at the heart of the problem instead of trying to patch around it. By trying to eliminate timing attacks people have introduced other timing attacks in their code as shown by Coda Hale. (C. Hale, 2009)

Timing attacks can be as dangerous as buffer overflows, format string exploits, and any other attack, which gives attackers access to a server. For example, a timing attack in OpenSSH could allow an attacker to gain access to a server.

# Appendix A – Python Client Code

The following is our python client that performed the timing attack

```
import socket
import sys
import time
import csv
from collections import defaultdict

HOST = 'localhost' # The remote host
PORT = 31331 # The same port as used by the server
PASSWORD = ['a','a','a','a','a','a','a','a','a']
TRIES = 100
CURRENT_CHAR = 0

timings = {}
def update_password():
    yield PASSWORD[CURRENT_CHAR]
    while PASSWORD[CURRENT_CHAR] != 'z':
        new_char = chr(ord(PASSWORD[CURRENT_CHAR]) + 1)
        PASSWORD[CURRENT_CHAR] = new_char
        yield new_char

def get_best_char():
    max = 0
    best = 'a'
    for key, value in timings.iteritems():
        if value > max:
            max = value
            best = key
    return best

def get_bottom_tenth(times):
    times = sorted(times)
    index = int(len(times) * 0.95)
    print 'Grabbing at %d with value %f' % (index, times[index])
    return times[index]

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
```

```

output = open('output-timing.csv', 'w')
writer = csv.writer(output)
for x in xrange(0,8):
    CURRENT_CHAR = x
    timings = {}
    run_times = defaultdict(list)
    for tries in xrange(0, TRIES):
        for each_char in update_password():
            password = ".join(PASSWORD)
            start = time.time()
            s.send(password)
            data = s.recv(1024)
            stop = time.time()
            run_times[each_char].append((stop - start))
            if data == 'Password Correct':
                raise Exception('Found password %s' %
password)
        PASSWORD[CURRENT_CHAR] = 'a'

    for char, time_list in run_times.iteritems():
        sorted_list = sorted(time_list)
        timings[char] = get_bottom_tenth(sorted_list)
        row = [char,]
        row.extend(sorted_list)
        writer.writerow(row)

    print 'Run times %s' % timings
    best = get_best_char()
    print 'Best %s' % best
    PASSWORD[CURRENT_CHAR] = best
    time.sleep(2)
s.close()
print 'Received', repr(data)

```

# Appendix B - Python Server Code

The following is our python server that we attacked against.

```
#!/usr/bin/python
"""
Authentication server program written by Evrhet Milam
"""
import socket
import time

HOST = ""          # Symbolic name meaning all available interfaces
PORT = 31331      # Arbitrary non-privileged port. Elite Hax0r
PASSWORD = 'xxelitex'

def check_password(user_input):
    user_input = user_input.strip()
    for x, each_char in enumerate(PASSWORD, start=0):
        if each_char != user_input[x]:
            return False
    return True

def listen():
    print 'Listening for connections on port %d' % PORT
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    print 'Connected by', addr
    while 1:
        data = conn.recv(1024)
        if not data: break
        if check_password(data):
            conn.send('Password Correct')
        else:
            conn.send('Password Incorrect')
    conn.close()

if __name__ == '__main__':
    while True:
        listen()
```

# References

---

<sup>i</sup> Paul C Kocher, Joshua Jaffe, and Benjamin Jun, “Introduction to Differential Power Analysis and Related Attacks” (2009)

[http://www.cryptography.com/public/pdf/DPA\\_TechInfo.pdf](http://www.cryptography.com/public/pdf/DPA_TechInfo.pdf)

<sup>ii</sup> Nate Lawson, “Timing Attack in Google Keyczar library” (2009)

<http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/>

<sup>iii</sup> Coda Hale, “A Lesson In Timing Attacks”, <http://codahale.com/a-lesson-in-timing-attacks/>

<sup>iv</sup> D. Boneh and Brumley, “Remote Timing Attacks are Practical” (2003)

<http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf>

<sup>v</sup> Crosby, Wallach, and Riedi, “Opportunities and Limits of Remote Timing Attacks” ACM Transactions on Information and System Security (TISSEC)

TISSEC Volume 12 Issue 3, January 2009

<sup>vi</sup> H. Bar-El, “Introduction to Side Channel Attacks”

<http://www.discretix.com/PDF/Introduction%20to%20Side%20Channel%20Attacks.pdf>