Birju Patel

Dr. Garrison

February 28, 2024

Project 1 Report

Summary

To execute Paul Kocher's timing attack on RSA, the adversary exploits the fact that the runtime of the modular exponentiation algorithm varies depending on the input given to it. To discover the hidden input value, an adversary must take many precise measurements of the runtime. It is often assumed that the adversary has physical access to the machine running the algorithm, and can thus take these measurements easily using an OS level facility such as the Linux time command. However, the adversary may attempt to execute this attack remotely by sending requests to a server that runs a service which is publicly exposed to the internet, and then measuring the amount of time it takes for them to receive a response. Such an attack is hard to detect, because it would be difficult for the system administrator to tell the difference between the adversary's requests and legitimate user traffic.

By expanding the scope of the attack, the adversary also introduces new variables. The adversary experiences latency when sending and receiving messages from the server. This delay is totally uncorrelated with the runtime of the algorithm, and is thus a source of noise that could potentially drown out any information that could be gleaned from the timing measurements. This problem can be partially addressed by utilizing the fact that TCP packets are timestamped as they are sent. By inspecting these timestamps, we can get a less noisy measurement of when the server is finished its calculation. However, this timestamp is not completely reliable. All clocks on all computers are slightly out of sync and run at slightly different rates, so the timestamps need to be corrected to reflect the difference between the time on the server and the time on the adversary's machine.

A fully remote attack is likely impossible, as latency over the internet varies widely between 10 and 100 milliseconds, while the runtime of modular exponentiation varies on the order of microseconds. To make the attack easier, I assume that the adversary's machine and the targeted server are on the same LAN. According to Spreitzer's taxonomy of side-channel attacks, this would make the proposed attack is an active, logical, and vicinity attack. The goal of this project is to determine if such an attack is feasible.

Experimental Setup

According to various forum posts, the average clock skew between computers on a LAN that uses Network Time Protocol (NTP) for clock synchronization is below 10 milliseconds, and

latency is usually below 5 milliseconds. For this experiment, I assume clock skew is fixed at 7.5 milliseconds, and that latency averages 3 milliseconds and varies by a standard deviation of 1 millisecond.

To simulate the attack, I created a simple client that interacts with a vulnerable server. The client sends the server a string that contains the hexadecimal representation of a large integer. The server then uses a private constant (x) and a public constant (n) to run the modular exponentiation algorithm on the value (y) provided by the client. The server uses a naïve modular exponentiation algorithm that makes no attempt to make its runtime uniform, and is thus vulnerable to a timing attack. The server then sends a message back to the client containing the string representation of the result (y^x mod n) of that calculation and a timestamp recording when that message was sent. For this experiment, I assume that y and n are 64-bit values, and x is a 32-bit value. The client can also ping the server. Upon receiving a ping, the server will send a message containing only the timestamp immediately back to the client, without completing any calculation. To simulate latency, I have the server wait for a random amount of time after it receives a request from the client and before it sends a response to the client. To simulate clock skew, I add 7.5 milliseconds to every timestamp before sending the response.

The portion of the RSA algorithm that is vulnerable to a timing attack is the modular exponentiation step. If an adversary can discover the private constant x, they can also discover all the other information they need to break the security of RSA. Therefore, this simplified setup is sufficient to demonstrate the feasibility of the attack.

Data Collection Procedure

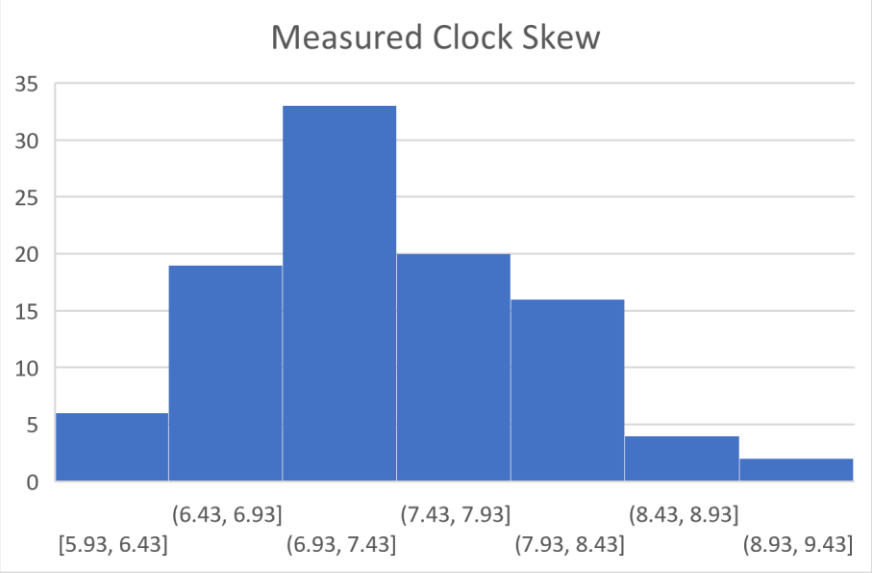
To measure clock skew, I measure the total time it takes to complete a ping on the client side (rtt), and I also record the time the client sent the ping (t0). On average, the latency to and from the server is the same. Therefore, we expect that the ping reached the server at t0 + 0.5 * rtt. This is what we expect the timestamp (ts) to read. If there is a difference, we know there was clock skew. We can therefore approximate clock skew as ts – (t0 + 0.5 * rtt). We can repeat this several times and take the average to get a better approximation. I then use this clock skew measurement to correct future timestamps.

To determine if a client's current guess of the value of x is more correct than its previous guess, I measure the variance in computation time. The client first records the time, then generates a random y value and sends it to the server. When the client receives the response, the difference between the start time and the timestamp of the response minus the clock skew is recorded. We assume that the client has guessed up to b bits of x. The client now runs a modified version of the modular exponentiation algorithm, where the algorithm is run for only the first b bits of x. The time this computation takes is then recorded. The local calculation time is subtracted from the server's calculation time to yield the timing difference. This experiment is then repeated many times. The variance of the timing difference is calculated. Theoretically, a lower observed variance should indicate a more accurate guess. The client program prints whatever data it measures directly to the console, and I save this output to a file for analysis.

The client can be configurable to take either low precision millisecond measurements or high precision nanosecond measurements. It is possible for TCP timestamps to be configured such that they are recorded with nanosecond precision. However, in real life, encountering a system configured like this is unlikely, so the adversary will likely only have access to millisecond timestamp values. For the purposes of the experiment, I measure the time in nanoseconds.

Results

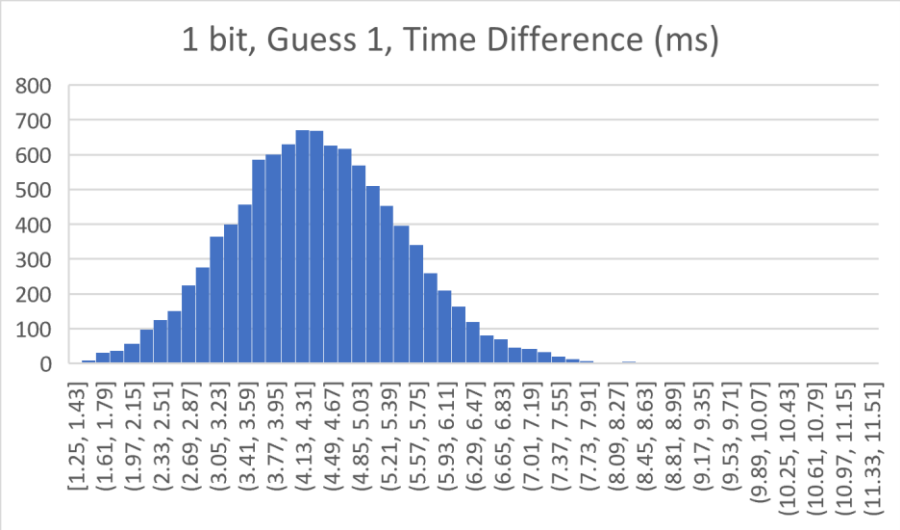
The protocol to measure clock skew is very accurate. Below are the clock skew measurements from 100 pings. During this experiment, the client measured an average clock skew of 7.402722 milliseconds, which is remarkably close to the true value of 7.5 milliseconds. For the next experiments, before calculating timing variance, we first calibrate our estimate of clock skew by pinging the server 250 times.

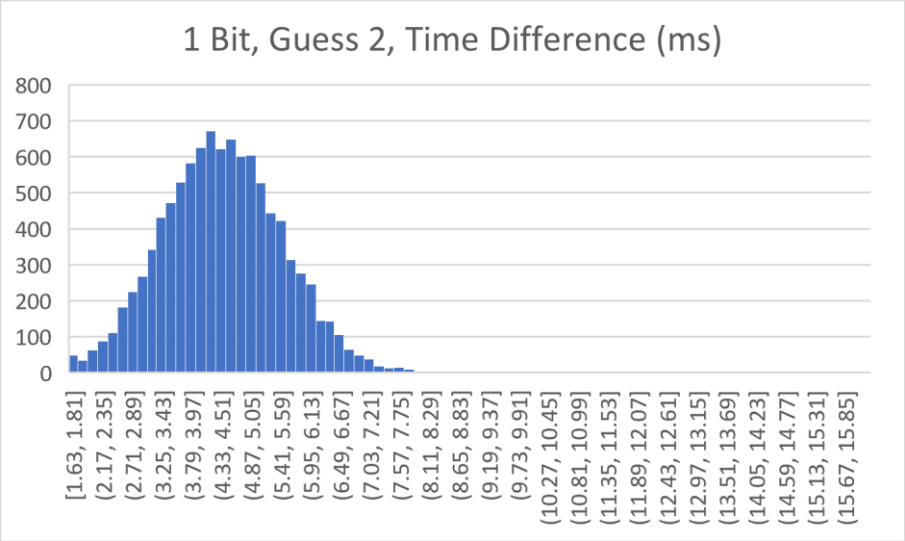


I then measured the timing difference for various pairs of correct and an incorrect guess of the first b bits of x. I considered the experiment successful if the variance of the time for the correct guess was small enough compared to the variance of the time for the incorrect guess that the difference was considered statistically significant using a p-value of 0.1. In one experiment, I was able to infer the value of the first bit of x from the distributions using this test. In another, I was able to infer the sixteenth bit. Below are the results from these two experiments.

Experiment 1	
Guess 1 (Correct)	0x0
Guess 2 (Incorrect)	0x1
x	0x14d2be72
Guessed Bits	1
Number of Samples	10000

Variance(1)	1.182068609
Variance(2)	1.21350072

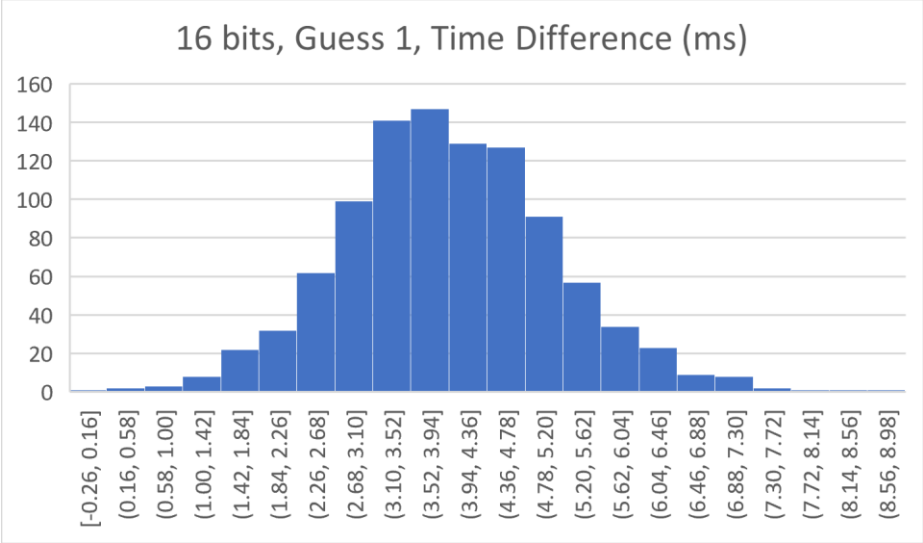


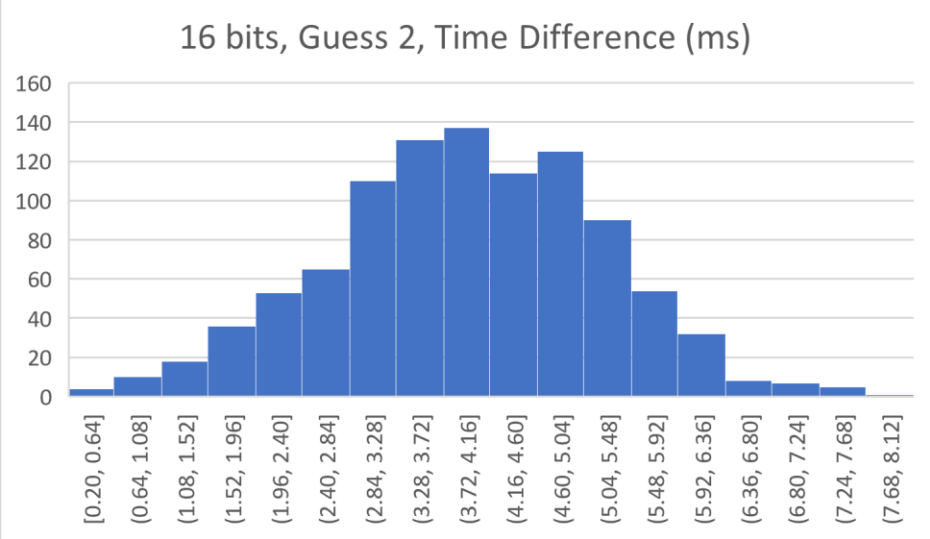


F-Test Two-Sample for Variances		
	Guess 1	*Guess 2*
Mean	4.413629	4.447876
Variance	1.182069	1.213501
Observations	10000	10000
df	9999	9999
F	0.974098	
P(F<=f) one-tail	0.094751	
F Critical one-tail	0.974692	

Experiment 2

Guess 1 (Correct)	0xbe72
Guess 2 (Incorrect)	0x3e72
x	0x14d2be72
Guessed Bits	16
Number of Samples	1000
Variance(1)	1.428547289
Variance(2)	1.599805084





F-Test Two-Sample for Variances		
	Guess 1	*Guess 2*
Mean	3.944662	3.944926
Variance	1.428547	1.599805
Observations	1000	1000
df	999	999
F	0.892951	
P(F<=f) one-tail	0.036855	
F Critical one-tail	0.922079	

Conclusion

In both experiments, the differences in variances are statistically significant, as indicated by the fact that the measured F-statistic is below the F critical value. Since it is possible to detect this difference by observing only the timing data, it becomes possible for the client to infer the value of the server's hidden value x bit by bit. Since the security of the RSA encryption scheme relies on this value remaining hidden, the following method could be used to compromise its security. Doing so would allow the adversary to decrypt encrypted messages sent to the server, or if the server is being used as a decryption oracle, allow the adversary to impersonate the server by replicating its digital signature. By demonstrating that a remote adversary can still infer a portion of this value despite the error introduced by the network latency and clock skew, this project has succeeded in its goal of showing the feasibility of running the timing attack remotely.

Future Work

This project could be extended by increasing the simulated latency and clock drift from beyond what is expected on a LAN and towards what is seen on the open internet. Doing so would expand the scope of the attack from a vicinity attack to a fully remote attack. This could be accomplished with my existing setup by increasing the simulated latency and clock drift, and by programming the clock drift to change over time. Due to time constraints, I was only able to run experiments that used small 64-bit messages and small 32-bit keys. A real-world RSA implementation would use 256-byte messages and a 1024-bit key. When measuring the computation time and clock skew, I used a timer with nanosecond precision. While this allowed me to get precise measurements, it did not reflect real-world conditions, as TCP timestamps are usually only configured to provide the time with millisecond precision. In a future experiment, one could determine if the attack is feasible if the adversary only has access to a clock with millisecond precision.

Reflection

I greatly enjoyed working on this project. Coming into this semester, I already had a good deal of experience with network programming from other courses, so it was rewarding to be able to leverage that knowledge during this project. During my initial reflection, I stated that one of my course goals was to design and implement security technology. Through the course of this project, I was able to do that. I also learned how to look at the design of a piece of technology and find vulnerabilities that could possibly leak private information.

Notes

To compile, navigate to */app* and run *javac -cp .;* *.java*.

To run the server, the command is *java Server [port]*.

To run the client, the command is *java Client [serverIP]:[serverPort]*.

The generator allows you to generate random x and n values. Run *java Generator* and these values will be printed as a hexadecimal number.

To change the experiment setup, you must manually change flags and values in the *Client.java* file and recompile. When the client is run, it will automatically run the configured experiment and print the data to the console in the format *sample number, time*. Below is a list of the options and their meanings.

guessX – A correctly guessed portion of the hidden value x.

wrongGuessX – An icorrectly guessed portion of the hidden value x.

guessBits – The number of bits b that have been guessed.

useWrongGuess – When true, run experiment with wrongGuessX. When false, run experiment with guessX.

EXPERIMENT_TYPE – Whether to measure only clock skew (0) or measure clock skew and then timing difference (1).

SAMPLES – Number of samples to collect.

NUMPINGS – During a type 1 experiment, the number of times the server is pinged to calibrate the clock skew measurement.

NANO – When true, prints the timing measurements in nanoseconds. When false, prints the timing measurements in milliseconds.

For part of this project, I reused code that I previously wrote for Dr. Farnan's course CS1501. All the reused code is contained in the file *HeftyInteger.java*.