**Design**

Question #1

overtData <- read.csv(file = "Traffic\_data\_orig.csv", header = TRUE,sep = ",")

secretMessage <- "this is a secret message"

messageLen = as.numeric(nchar(secretMessage))

CharToBinary = function(pchar) {

lhex = charToRaw(pchar)

lbits = rev(as.numeric(rawToBits(lhex)))

return(lbits)

}#Converts a character to Binary equivalent

StringToBinary = function(pstr, pstrlen) {

lbitstream = NULL

ltemp = NULL

for (i in 1:pstrlen) {

ltemp = CharToBinary(substring(pstr,i,i))

lbitstream <- c(lbitstream,ltemp)

}

return(lbitstream)

}#converts a string to Binary vector

#Q1. GENERATING MODIFIED PACKET STREAM

binaryMessage = StringToBinary(secretMessage, messageLen)

lenBinaryMessage = as.numeric(length(binaryMessage))

covertPackets = NULL

timeStream = 0

covertDataDelay = NULL

for (i in 1:lenBinaryMessage) {

if(binaryMessage[i] == 0){

timeStream <- timeStream + 0.25

covertDataDelay <- c(covertDataDelay,0.25)

}#if bit is 0, delay is 0.25

else{

timeStream <- timeStream + 0.75

covertDataDelay <- c(covertDataDelay,0.75)

}#if bit is 1, delay is 0.75

covertPackets <- c(covertPackets,timeStream)

}#loop to generate packet stream from binary message

#Q2. Generating the Histogram

temp = 0 #holds the previous timestamp

overtDataDelay = numeric(dim(overtData)[1])

for( i in 1:dim(overtData)[1])

{

overtDataDelay[i] = overtData[i,2] - temp

temp = overtData[i,2]

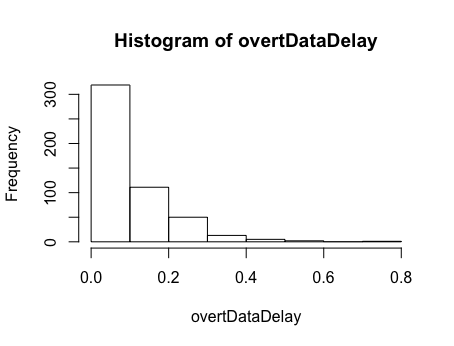
}

hist(overtDataDelay)

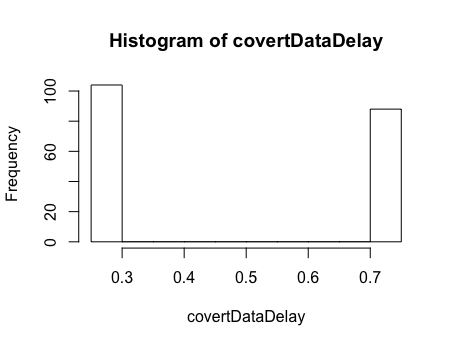
hist(covertDataDelay)

Question #2

**Overt Packet Delay Histogram**

****

**Covert Packet Delay Histogram**



Will Eve be suspicious?

Eve will be suspicious of the 0.75 & 0.25 because it's exactly two unique differences, whereas for the "Overt" data, given to us in the CSV, the histogram looks more like a negative exponential distribution, which is what you would expect of this type of time dependent data

Question #3

#Q3

m = median(overtDataDelay)

min = min(overtDataDelay)

max = max(overtDataDelay)

semiRandomPacketsDelay = NULL

semiRandomPackets = NULL

timeStream = 0

for (i in 1:lenBinaryMessage) {

if(binaryMessage[i] == 0){

timeLapse = runif(1, min, m)

}#if bit is 0, delay is 0.25

else{

timeLapse = runif(1, m, max)

}#if bit is 1, delay is 0.75

timeStream <- timeStream + timeLapse

semiRandomPacketsDelay <- c(semiRandomPacketsDelay,timeLapse)

semiRandomPackets <- c(semiRandomPackets,timeStream)

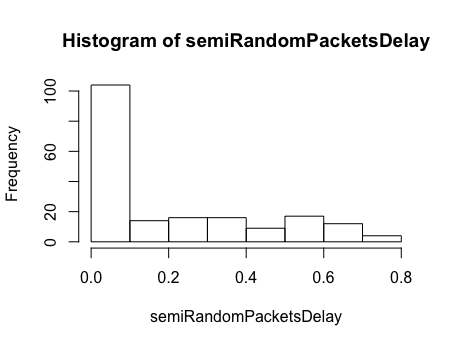
}#loop to generate packet stream from binary message

hist(semiRandomPacketsDelay)

hist(overtDataDelay)

**Note: Code is built up on code from question 1**

Question #4

****

Question #5

1. **Improved Method**

Since we know that the overt timing delay dataset produces an exponential distribution (pdf), we need to make the covert dataset look as exponential as possible so Eve doesn’t suspect us.

We can use the mean m of the overt data set to generate an exponential distribution (using rexp).

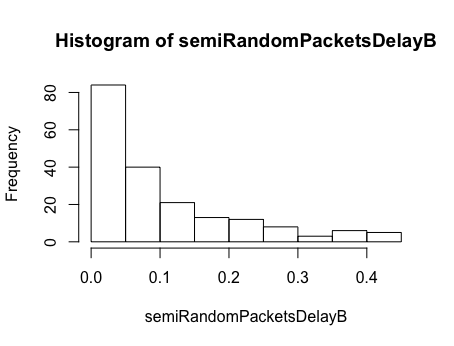
Alice can base the covert delay times on the generated exponential distribution with some alterations to specify the bit values. The cutoffs for 1s and 0s would be represented as a function of the packet number (the 1st packet of the two packets that the delay is between).

**Encoding Scheme:-**

* + Exp(i) - 0.0025 for bit = 1; where i is the packet number and Exp(i) is the density that corresponds to i in an exp(rate = m^-1)
  + Exp(i) + 0.0025 for bit = 0; where i is the packet number and Exp(i) is the density that corresponds to i in an exp(rate = m^-1)

**Assumptions:-**

* Bob knows the distribution of the overt timing dataset (exponential with rate = m)
* Bob knows the encoding scheme so he can decode the message.

****

**Code**

set.seed(1237)

#m = median(overtDataDelay)

#m = quantile(overtDataDelay, probs = c(0.75) ,names= FALSE)

m = mean(overtDataDelay)

min = min(overtDataDelay)

max = max(overtDataDelay)

semiRandomPacketsDelayB = NULL

semiRandomPacketsB = NULL

timeStream = 0

Nexp <- rexp(1000000000000000, m^-1)

for (i in 1:lenBinaryMessage) {

if(binaryMessage[i] == 0){

timelapse = Nexp[i] - 0.0025

#timeLapse = rexp(1, m)

}#if bit is 0, delay is 0.25

else{

timeLapse = Nexp[i] + 0.0025

#timeLapse = runif(1, m)

}#if bit is 1, delay is 0.75

timeStream <- timeStream + timeLapse

semiRandomPacketsDelayB <- c(semiRandomPacketsDelayB,timeLapse)

semiRandomPacketsB <- c(semiRandomPacketsB,timeStream)

}#loop to generate packet stream from binary message

hist(semiRandomPacketsDelayB)

1. If Alice buffers the packets, there will be a moment (in the beginning) when no packets are sent which would disrupt the continuity of the timing channel; this is obviously unrealistic in the case of a Skype call, which is based on real-time interaction. Overall, it would negatively affect the quality of the call, leading Eve to suspicion.
2. If the network modifies the inter packet delay, the message cannot be decoded correctly since the delays are changed and each bit is encoded via delay times. If Bob knows the distribution of the noise within the channel, he can subtract the mean of the noise’s distribution from the covert delay times while decoding to get a somewhat accurate message. This method is obviously flawed as it assumes that the semantic integrity of the message isn’t critical.

**Detection**

Step #1

*Code*

*x <- rnorm(30,mean = 0,sd = 1)*

*y <- rnorm(30,mean = 0,sd = 1)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

Step #2

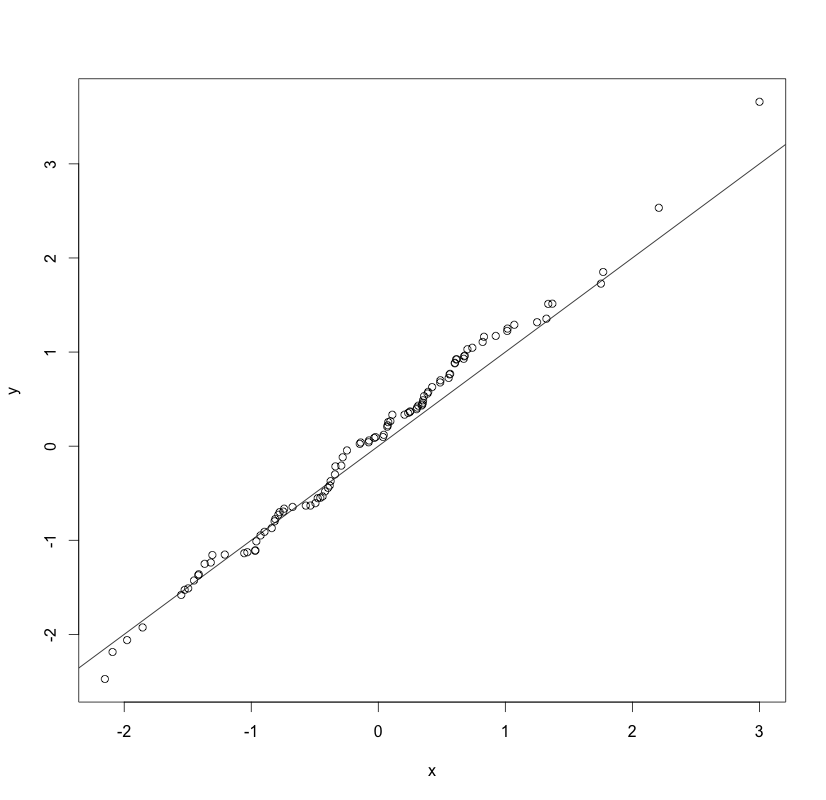
*Code*

*x <- rnorm(100,mean = 0,sd = 1)*

*y <- rnorm(100,mean = 0,sd = 1)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

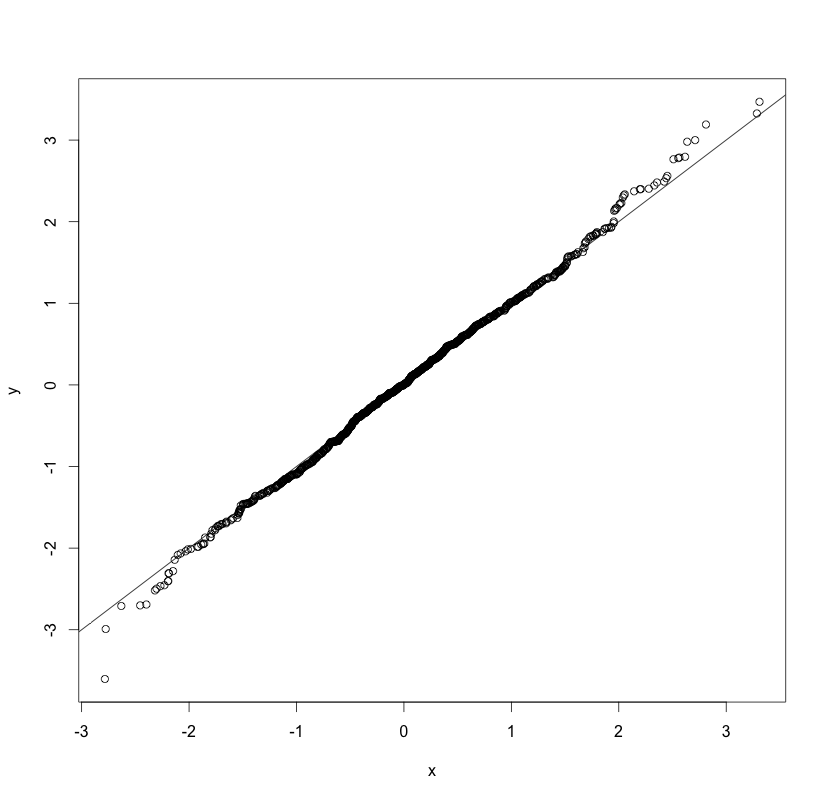
**

*x <- rnorm(1000,mean = 0,sd = 1)*

*y <- rnorm(1000,mean = 0,sd = 1)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

**

*Observation*

#It is clear that the shape of the qqplot becomes more linear as n increases.

#This makes sense, as the two distributions are exactly the same.

#The points are dense on the graph between the 1st and 3rd quartiles; they are especially dense around the mean.

Step #3

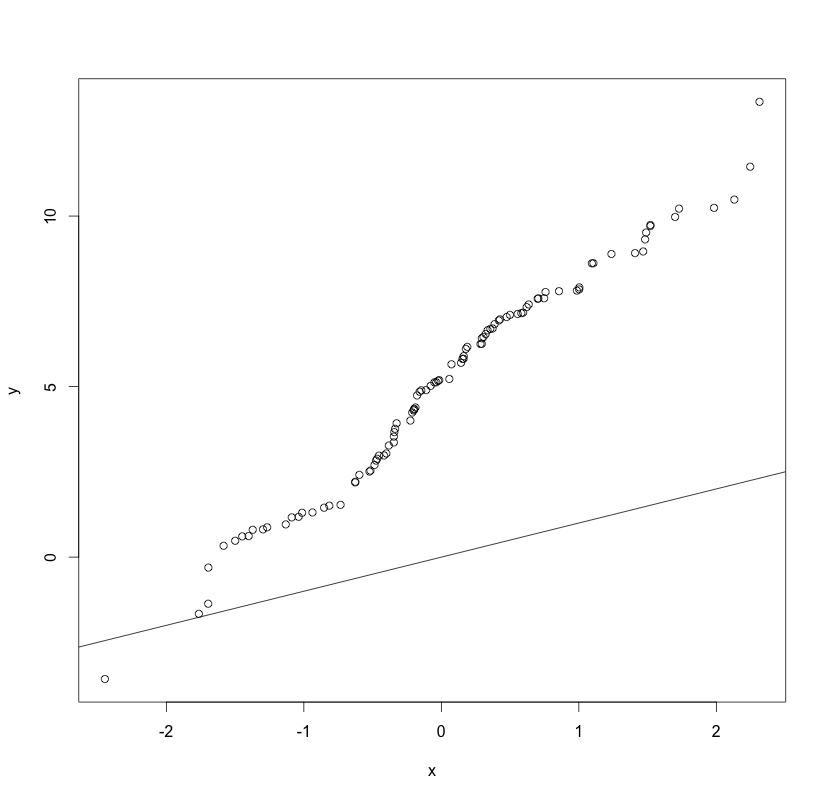
*Code*

*x <- rnorm(100,mean = 0,sd = 1)*

*y <- rnorm(100,mean = 5,sd = 3)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

**

*Observation*

The shape of this qqplot also appears to be linear, however the slope is much larger than 1 (which would have indicated perfect correlation). Also, the line itself is shifted up so that at x = 0, y = 5. This shows the relationship between the means: the mean of x is 0, and the mean of y is 5. Using this information, we can apply similar rational to the difference in slope. It is caused by the different standard deviations. The slope for y vs x is 3 in our graph, which also describes the ratio of the standard deviations between the two data sets. Despite the fact that the means and standard deviations are different, and that the numerical values for slope and intercepts change because of it, if we normalize y with respect to x, we see that the it does indeed fall on the slope=1 line.

Step #4

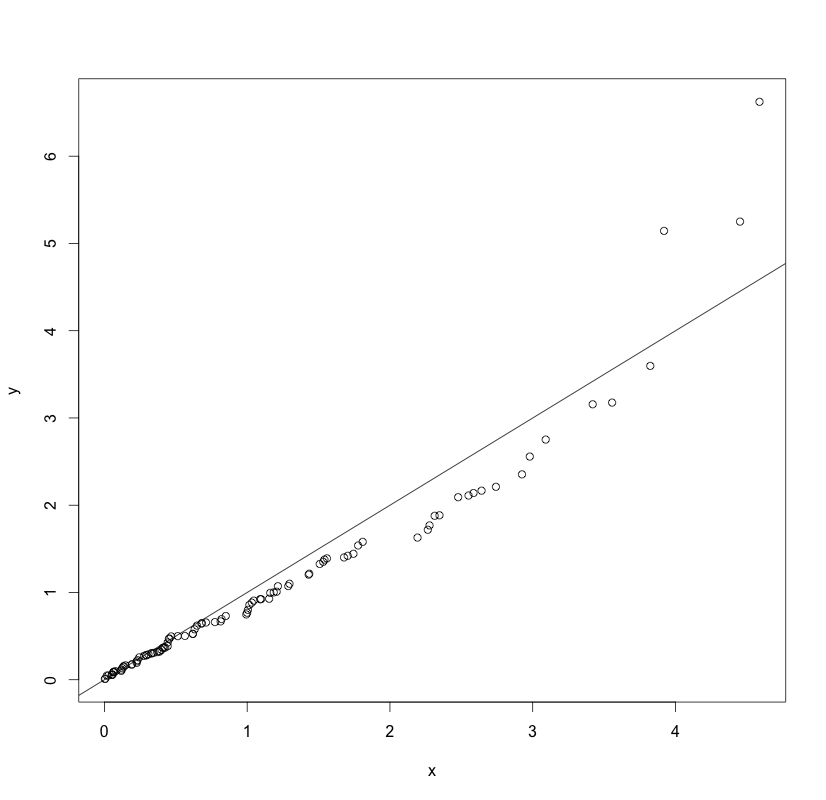
*Code*

*x <- rexp(100,rate = 1)*

*y <- rexp(100,rate = 1)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

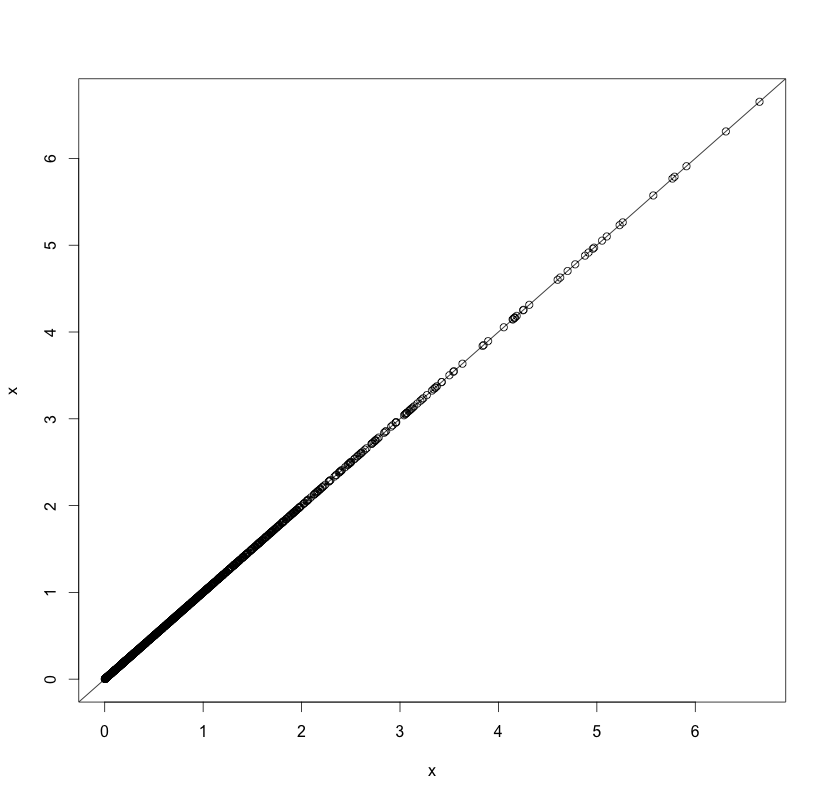
**

*x <- rexp(1000,rate = 1)*

*y <- rexp(1000,rate = 1)*

*qqplot(x,y, plot.it = TRUE)*

*abline(0,1)*

**

*Observation*

#It is clear that the shape of the qqplot becomes more linear as n increases.

#This makes sense as the two distributions are exactly the same.

#The points are dense around 1 which makes sense as the mean for an exponencial distribution is (lamda)^-1.

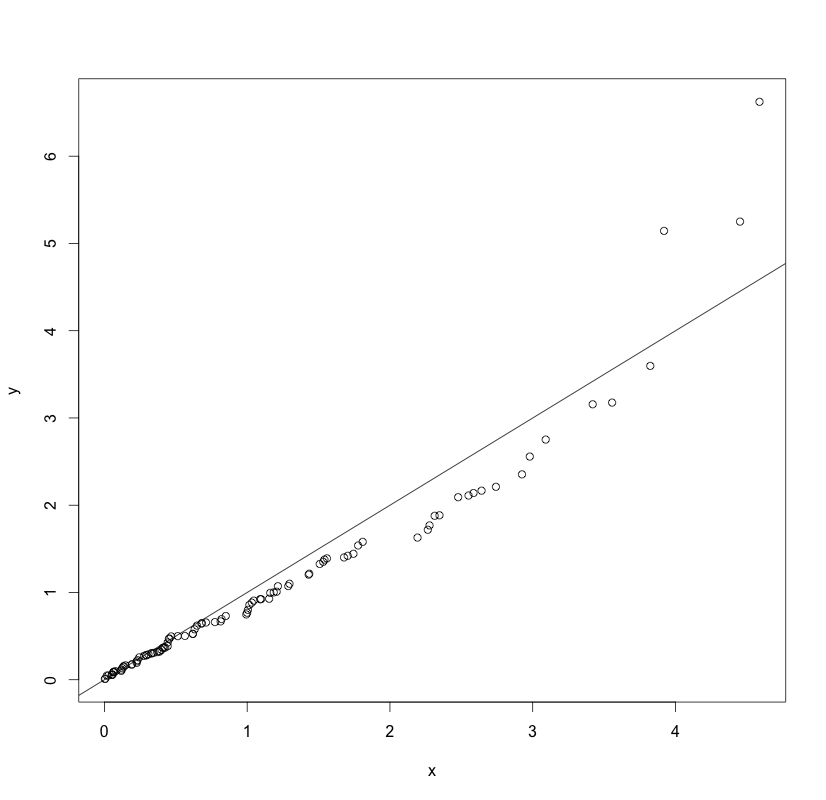
Step #5

*Code*

x <- rnorm(100,mean = 0,sd = 1)

y <- rexp(100,rate = 1)

qqplot(x,y, plot.it = TRUE)

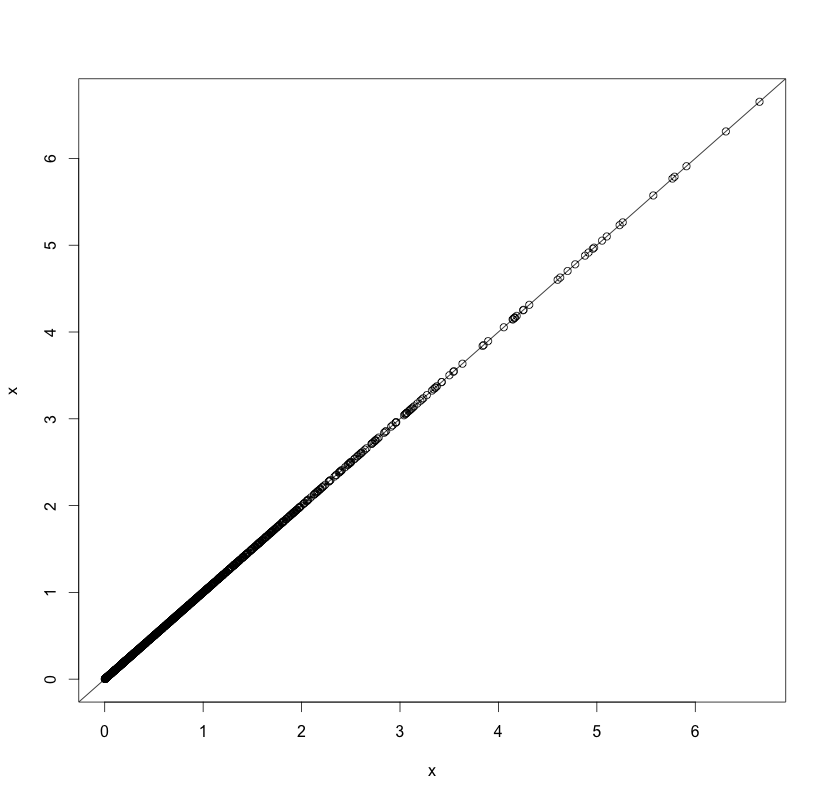


x <- rnorm(500,mean = 0,sd = 1)

y <- rexp(500,rate = 1)

qqplot(x,y, plot.it = TRUE)

abline(0,1)

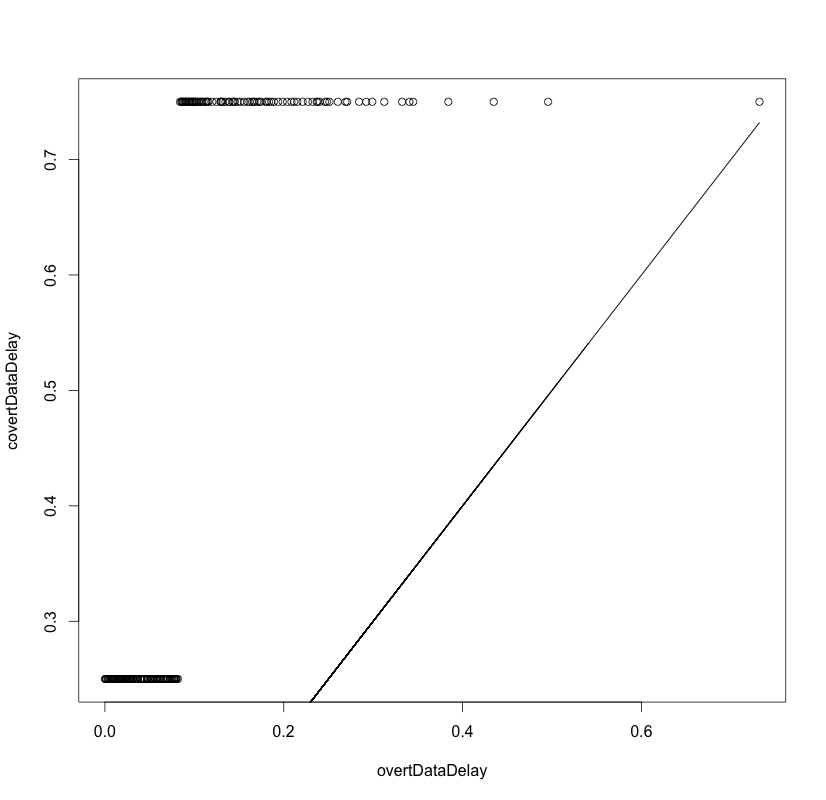


Step #6

*Code*

*qqplot(overtDataDelay, covertDataDelay, plot.it = TRUE)*

*lines(overtDataDelay, overtDataDelay)*

**

*Observation*

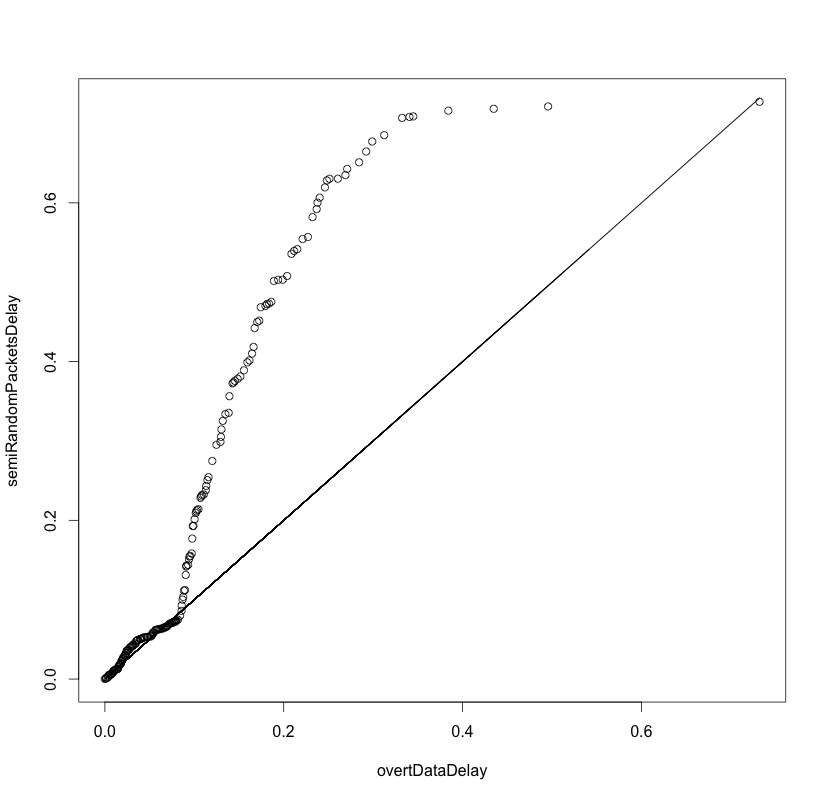
Clearly, the two distributions are very different and easily distinguishable from each other because the points are nowhere near the slope 1 line. Therefore, Eve will easily detect messages.

Step #7

*Code*

*qqplot(overtDataDelay, semiRandomPacketsDelay, plot.it = TRUE)*

*lines(overtDataDelay,overtDataDelay)*



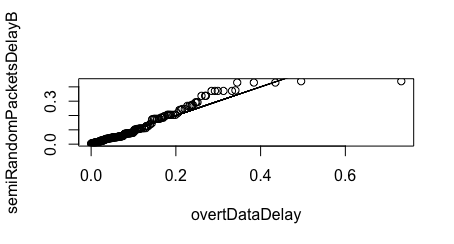
*Observation*

Although this distribution initially follows closely along the exponential, it quickly diverges. Therefore, Eve will also easily detect this distribution as well. However, it is still better than the method in Step #6

Step #8

*Code*

qqplot(overtDataDelay, semiRandomPacketsDelayB, plot.it = TRUE)



*Observation*

Of the distributions we have developed and tested, this appears to be the best. Eve will probably still be suspicious, but significanly less so and may even careless attribute this to noise, some error in the network, or other confounding variables.

**Note: Code is built up on code from question 1**

**Implementation**

1. IPD is exponential with λ = 1

|  |  |  |  |
| --- | --- | --- | --- |
| m | i | P(overflow) | P(underflow) |
| 16 | 2 | 0 | 0.561 |
|  | 6 | 0 | 0.43 |
|  | 10 | 0 | 0.325 |
|  | 14 | 0 | 0.199 |
|  | 18 | 0 | 0 |
| 32 | 2 | 0.023 | 0.462 |
|  | 6 | 0.017 | 0.397 |
|  | 10 | 0.022 | 0.32 |
|  | 14 | 0.019 | 0.216 |
|  | 18 | 0.02 | 0.203 |

1. IPD is uniform

|  |  |  |  |
| --- | --- | --- | --- |
| m | i | P(overflow) | P(underflow) |
| 16 | 2 | 0 | 0.591 |
|  | 6 | 0 | 0.565 |
|  | 10 | 0 | 0.514 |
|  | 14 | 0 | 0.37 |
|  | 18 | 0 | 0 |
| 32 | 2 | 0 | 0.76 |
|  | 6 | 0 | 0.726 |
|  | 10 | 0 | 0.657 |
|  | 14 | 0.019 | 0.631 |
|  | 18 | 0.02 | 0.584 |

1. Bounds on Overflow & Underflow Probabilities

Event Definitions:-

i : The number of packets currently in the buffer.

B = Buffer size

P: Overflow:-

* When i = B and the source sends another packet, the source packets cannot be buffered anymore

Q: Underflow:-

* When i = 0 and sender needs to send another packet, the sender cannot send any more packets

A round:-

* Out: If a packet is sent by sender -> i = i – 1 [Gamblers Ruin – “Lose”]
* In: If a packet is received by sender -> i = i + 1 [Gamblers Ruin – “Win”]

Assumptions:-

* Each round is independent from the other rounds
* Packets being sent and received follow same distribution
* Gambler’s ruin formula holds

**From Gambler’s Ruin Formula:-**

Let p = P(In) ; q = P(Out)

*See attached binder paper and the end for complete answer with bounds for overflow and underflow*

1. Propose methods to deal with buffer overﬂow and underﬂow.

**Dealing with Overflow**

* Incorporate a secondary buffer :
  + If the primary butter is full, the source can put packets into the secondary buffer.
* Increase buffer size, B:
  + This would directly decrease the probability of having an overflow as more packets can be accommodated in the buffer

**Dealing with Underflow**

* Increase i i.e. required # of packets before starting to process
  + This would directly decrease the probability od having an underflow as it forces more packets to be buffered before the sender can send them. So, the sender is less likely to run out of packets while processing.

**Dealing with Overflow and Underflow**

* Increase B and i (See why above)

**Part 5 Code (some functionality build upon previous sections’ code)**

underOrOver = function(m, i, d){

#Buffer vars

B = 20

possible = c(0,1)

binaryMessage = sample(possible, m , replace = TRUE) #Random message of size m

if (d == 0){

generatePacketDelays = rexp(m, rate = 1) #Random sequencce of packets with exp distribution

}

else if(d == 1){

generatePacketDelays = runif(m, 0, 1) #Random sequencce of packets with unif distribution

}

timePacketStream = 0

timeLapse = 0

for(index in 1:m){

timeLapse = timeLapse + generatePacketDelays[index]

timePacketStream <- c(timePacketStream, timeLapse)

}

timePacketsStream = round(timePacketStream, digits = 3)

#Simulation for exp dataset

currentTime = 0

m = median(generatePacketDelays) #median of exp dataset

min = min(generatePacketDelays) #min of exp dataset

max = max(generatePacketDelays) #max of exp dataset

semiRandomPacketsDelay = numeric(m+1)

semiRandomPackets = numeric(m+1)

senderTimeStream = 0

senderPackets = NULL

senderPacketsDelay = numeric(m+1)

Delay = 0

start = FALSE

sourcePacketIndex = 1

senderMessageIndex = 1

numUnderFlow = 0 #numeric(1000)

numOverFlow = 0 #numeric(1000)

CB = 0

Buffer = NULL

for(bit in binaryMessage){

if(bit == 0){

Delay = runif(1,min,m)

}#if bit is 0

else{

Delay = runif(1,m,max)

}#if bit is 1

senderTimeStream = senderTimeStream + Delay

senderPackets <- c(senderPackets, senderTimeStream)

senderPacketsDelay <- c(senderPacketsDelay, Delay)

}#Precalculates sender delay times

senderPackets = round(senderPackets, digits = 3)

#print(paste("PRE-LOOP BUFFER"), Buffer)

while(length(binaryMessage) > 0 && sourcePacketIndex < length(timePacketsStream)){

if(currentTime >= timePacketStream[sourcePacketIndex] && CB < B){

#print(paste("ADD T0 BUFFER", currentTime))

Buffer <- c(Buffer, sourcePacketIndex)

CB <- length(Buffer)

sourcePacketIndex = sourcePacketIndex + 1

}#change sourceindex when reaches currenttime and places source packet into buffer

else if(currentTime >= timePacketStream[sourcePacketIndex] && CB >= B) {

#print(paste("SKIP BUFFER", currentTime))

numOverFlow = numOverFlow + 1

#print(paste("OVERFLOW!!!", currentTime, numOverFlow))

break

}#if CB >= B --> overflow

if(CB == i && start == FALSE){

#print(paste("START!!", currentTime))

start= TRUE

}#if current buffer size is > i, sender can start sending packets

if((start == TRUE) && (CB > 0) && (senderPackets[senderMessageIndex] <= currentTime)){

#print(paste("POP FROM BUFFER!!", currentTime, senderPackets[senderMessageIndex], Buffer[1]))

CB = CB - 1 #buffer size decreases by 1

Buffer = Buffer[-1] #pop off from buffer

senderMessageIndex = senderMessageIndex + 1 #update message index

binaryMessage = binaryMessage[-1]

}#Sender sends a packet

else if((start == TRUE) && (CB == 0) && (senderPackets[senderMessageIndex] <= currentTime)){

numUnderFlow = numUnderFlow + 1

#print(paste("UNDERFLOW!!!" , currentTime))

break

}#if CB is empty, nothing to send so underflow

currentTime = currentTime + 0.001 #update current time

}

return(c(numOverFlow, numUnderFlow))

}

distribution = c(0,1) #0 for exponential, 1 for uniform

mArr = c(16,32)

i = c (2,6,10,14,18)

for(D in distribution){

if(D == 0){

print(paste("Exponential :"))

}

if(D ==1) {

print(paste("Uniform: "))

}

for (M in mArr){

print(paste(" M: " , M))

for (I in i){

print(paste(" I: ", I))

over = 0

under = 0

for(s in 1:1000){

#print(s)

A = underOrOver(M,I, D)

over = over + A[1]

under = under + A[2]

}

over = over/1000

under = under/1000

print(paste(" overflow probability: ", over))

print(paste(" underflow probability: ", under))

}

}

}