1) (a) let $c_i \equiv m_i^e \mod n \; \forall\, i \in 1,2,3$. Assuming n is large enough where collision between c_i, c_j doesn't happen, if we have a guess on what m could be, we can straight up compute all c_i and see which $c_i = c$ and then we know the original message. Note, that this doesn't mean we leaked the secret key d yet, just the original message is kinda leaked.

(b) Eve just need to repeat the same calculation 1000 times for each of ATTACK, HOLD, RETREAT, which is not a lot, only a total of 3000 calculation is needed to figure out the original message

(c) Let's get a random message from Z_n and call it m, compute $c' = m^e(c) \mod n$ and sent it to Bob. Since $c' \neq c$, Bob will be more than happy decrypt the content and sent it back, which means you will get mr as the result. from there, you can just do mr/m since you know m to get the original value r

2) (a) Attack will be successful, as Alice himself will fall for phishing (he will not check the URL) and the TLS protocol will receive a correct certificate with a CA that is trusted by Alice's browser. Even though that the TLS knows that Alice is not talking to www.comp.com, the protocal is still legit.

   (b) This will indeed not work, because during the verification step, the TLS process will fail as verification would return certificate for www.comp.com instead of https://www.comp.mallory.com.

   (c) This will not work, Even though DNS route to the right address with stolen certificate, the problem is that Mallory doesn't have access to the signing key of the original https://www.comp.com, which means Mallory may be able to pass the setting up step but cannot communicate with Alice properly afterward, thus useless.

3) (a) Alice seems to only sent to Cyan server, because whenever Alice transfer, cyan always get at least the amount of package that Alice transferred.

(b) first compute P_{bkg}, which is basically a normalized distribution of background messages. On round 1,2,3,5,6,8,10 Dave didn't send any package, and the vector addition become $[0, 0, 3] + [1, 0, 2] + [1, 1, 1] + [1, 1, 1] + [0, 0, 3] + [2, 0, 1] + [1, 0, 2] = [6, 2, 13]$ normalize by 21 you have

$$P_{bkg} = [0.2857, 0.0952, 0.6190]$$

Now we want to compute P_{Dave}. The only round where Dave is in are round 4, 6, 9, meaning the base vector is $[3, 0, 0] + [0, 1, 2] + [1, 0, 2] = [4, 1, 4]$, we know that the message sent by the background in these round is $(0 + 2 + 2) = 4$, and message by Dave is $(3 + 1 + 1) = 5$. The final equation become

$$P_{Dave} = \frac{1}{5}([4, 1, 4] - 4 * [0.2857, 0.0952, 0.6190]) = [\frac{4}{7}, \frac{13}{105}, \frac{32}{105}]$$

Seems like Dave talk to the Yellow server quiet a lot

(c) we already know that Alice only send message to Cyan server, the last question told us the Dave send message to either Yellow or Cyan at random, on round 2, we observed that only Carol and Alice were sending message and Yellow server received message, based on the assumption, it seems like Carol only send message to the Yellow server as well. since Magenta server receive message occasionally and no one but Bob is inspected, we already know one of the server Bob is choosing from is Magenta, on round 6, we observed 2 message from Alice and 1 message from Bob, but Cyan received 3 messages. This means, Bob has to be choosing randomly between the Magenta and Cyan server.

(d) one of the most obvious approach is to increase the threshold τ. so far $\tau = 3$ which is really low, but if latency is not an issue, higher τ would make it harder for adversaries to ear drop and generate a table over time.

Another way to create anonymity is to create end to end dummy traffic. For example, Even though Alice doesn't talk to any server but Cyan, maybe it can send a dummy message occasionally to other server to increase anonymity.

As an icing on top, why not trying other format of mix? In particular binomial-pool mixer is pretty cool too.

4) (a) well, since there is no upper limit on how you can use sum, you can just do

   i. SELECT SUM(Salary) FROM Employee WHERE TRUE
   ii. SELECT SUM(Salary) FROM Employee WHERE Name != "Alice"

   so you first get the entire sum, and then you get entire sum except Alice, meaning that you can get Alice by sum - (sum - ALice) = Alice

   (b) So we do the following query

   a) SELECT SUM(Salary) FROM Employee WHERE Name != "Alice" AND Type = "Full Time"
   b) SELECT SUM(Salary) FROM Employee WHERE Name != "Alice" AND Type = "Part Time"
   c) SELECT SUM(Salary) FROM Employee WHERE TRUE

   we know that query a, b are going through since type is approximately 50/50 for all entries, and we know query c is going through since it hits the entire table. As the result, we can do c - (a + b), as (a + b) = c - Alice, so we can get the salary for Alice in the end

   (c) The idea of the attack is just to count if Alice is over counted, using the following query

   a) SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary == 50000) OR Type = "Full Time"
   b) SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary == 50000) OR Type = "Part Time"
   c) SELECT COUNT(*) FROM Employee

   if we simply check if a+b result in c, in the case where it equals, it means Alice is not over counted by 1, thus the requirement (Name == "Alice" AND Salary == 50000) was never true, meaning the Alice's salary is not exactly 50000. If a+b = c+1, it means Alice got over counted, thus Alice's salary is exactly 50000

   (d) a) SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary >= 50000) OR Type = "Full Time"
   b) SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary >= 50000) OR Type = "Part Time"
   c) SELECT COUNT(*) FROM Employee

   Same idea previous, if a + b = c, then (Name == "Alice" AND Salary >= 50000) is never true, so Alice's salary is less than 50000. if a + b = c+1, then Alice's salary is greater or equal to 50000

   (e) So based on the previous question, we already have the condition to launch binary search on Alice's salary. Let the p_{left} be 10000, p_{right} be 500000, we can use binary search to narrow down the search toward the actual salary step by step similar to bonus of assignment 2. Out of the 3 queries, we first only have to perform the count all operation only once, so each iteration we perform a+b to check if solution is correct, in total, we have a upper bound of

$$\lfloor \log(490000) * 2 \rfloor + 1 = 39$$

   which is pretty nice upper bound already, but we can make it better.

   One thing that people don't realized is that rejection is also a source of information, that is, if you are rejected from making a request, and if you know it only fail on certain binary condition, it can still be used for binary search. Suppose the $p_{mid} = (p_{left} + p_{right})/2$ is the value you want to test, send the below query over

   i. SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary >= p_{mid}) OR Name != "Alice" in the case which server actually returned, you know that Alice is counted and it has exactly N entries, in the case where the query is rejected, you know there are exactly $N-1$ entries, thus the salary of Alice is less than p_{mid}

now we have this condition, we only need 1 query per search to determine if Alice's salary is to the left or right of the middle point. Thus our upper bound become

$$\lfloor \log(490000) \rfloor + 3 = 21 < 30$$

where the last 3 query are just the query to double check if Alice's query is exactly the value you would expected by using the equal queries in part c, since you might get 2 potential results.

Following is the equivalent of python $bisect_left$ equivalent on this attack. Let

$f(p_{mid})$ = SELECT COUNT(*) FROM Employee WHERE (Name == "Alice" AND Salary > p_{mid}) OR Name != "Alice"

Algorithm 1 Attack

```
$p_{left} \leftarrow 10000$
$p_{right} \leftarrow 500000$
**while** $p_{left} < p_{right}$ **do**
    $p_{mid} \leftarrow \lfloor (p_{left} + p_{right})/2 \rfloor$
    **if** $f(p_{mid})$ is **SUCCESS then**
        $p_{left} = p_{mid} + 1$ // since we know Alice's salary must be strictly greater than $p_{mid}$
    **else**
        $p_{right} = p_{mid}$
    **end if**
**end while**
**return** $p_{left}$
```

and if you want to ensure correctness, feel free to run a query to confirm the value is indeed p_{left} that we have shown can be done in at most 3 queries.