Part 3

We resourced various hash functions such as sha-1,sha-2,md4,md5. Since sha-1 was broken by google but would take an computational unfeasible time to break and sha-2 is the currently being used we decided to use md5 because it is the most straightforward to implement.

What is a hash function?
A hash function maps arbitrary strings,in our case numbers, to a fixed length output.
A hash function takes an input message of arbitrary length and produces a fixed length output.

What are the properties of a hash function?
Properties of a hash function -

1. Given a message M the output h(M) can be calculated very quickly.
2. Preimage resistant - Given an output Y it should be computationally infeasible to find M such that h(M) = Y
3. It should be strongly collision free. It should be computationally infeasible to find M1 and M2 such that h(M1) = h(M2). Given the large set of inputs possible there is likely and M1 and M2 that gives the same result but it should not be easily found by computers.

A good hash function is non reversible without having the key. It always maps to the same fixed size. Speed is a very important property depending on the use. In our case the hash function can't be too fast or attackers will have an easier job breaking our function.
If 1 byte is changed in the original then the whole hash should be completely different. Using this property hash functions can be used as an identity check. If you know the input and output then you can check if the data has been tampered with.

When the attacker gets a hashed password can he directly know what the hash function is? Or can he retrieve or investigate the hash?
When an attacker sees a hashed password they can't generally know what the hash function is off the top of their head unless it's incredibly simple. But they can use online resources to figure out the hash function. One simple method is by googling the hashed password. If the hash function used is older like md5 of sha1 then odds are google will display the original password or lead to a cracker. These 2 hash functions were previously used but nowadays they are considered broken. The internet has had years to crack and store the original password and the hashed password together in one table. Attackers can also narrow down their guess based on the size of the hashed password. Various hashing functions all generate a different sized output depending on the input. Since SHA -2 is the standard hackers can assume that SHA-2 is being implemented.

How does the attacker eventually obtain the hash so that the attacker can brute-force or even use a rainbow table to try any combination of characters through the hash in order to obtain the same hash value.

Security systems that use hash functions takes an input and stores the hash function from it. The system doesn't know the original password.
Attacker's can break into the system (website,database) to obtain the hash. They can access the hardware and steal the data using a usb.
After having the hashes the hacker just needs a passwords that generates the same hash. It doesn't matter if the password is the same as the original.
They can look through the rainbow table looking for that hash.

When to use a Dictionary attack vs. a Rainbow Table attack?
Dictionary attack is useful if you have information about who your attacking, such as names, dates etc.. A dictionary attack doesn't generate all possible passwords, instead it generates password combinations given the dictionary you chose reducing execution time. A dictionary attack will tend to use real world values instead of random values. Simple passwords made by humans are more likely to be picked up. A dictionary attack does not need the hash to attack. If you don't have access to the hash you can use a dictionary attack. A rainbow attack works in reverse. It needs the hash to find a corresponding password.

Rainbow Table attacks are useful in an offline environment or if you want to break multiple passwords you can reuse the table. The downside of the rainbow table is that all entries in the table must be created before you start because you don't know which will be the answer. If your 10th entry out of 1 billion entries is the match then the time needed to create entries after the 10th is wasted time. If you have no information about your target then dictionary attack is better. Rainbow Table's also require large amounts of storage, terabytes. If you don't have enough space to store the table you won't be able to use it. Dictionary attack does not do well with systems that store large passwords.

What are the resource requirements for each type of attack? Which uses more storage? Which requires more pre-computation?Which requires more analysis time? (per-hash vs. batch cracking)
There are rainbow tables available online. If one uses a pre existing rainbow table precomputation is not needed. If we have to make our own rainbow table than the pre-computation is immense and larger than dictionary attacks. A rainbow tables requires a large amount of hashes computed because we have no idea which hash to expect. The idea is to have such a large number of hashes that we get a matching hash. For this reason rainbow tables require more storage.
Dictionary attack take uses a dictionary of words to attack. It also has to make variations of the words in the dictionary, such as writing it backwards, which takes some computation time.

A rainbow table is larger than a dictionary table. A rainbow table needs to a large amount of hashes in order to increase the odds of finding a hash match. Since the amount of data is larger for a rainbow table the analysis time for a rainbow table is larger. The matching hash could be the 1st one on the table or the millionth.

To facilitate your understanding on the concept of assignment, let's consider a scenario for which thread 1 checks only combinations of length 1, thread 2 does that for 2 letter words, thread 3 for 3 letter words etc. So, you first generate the hashes and then matching them in bulk via your dedicated hardware (what type of HW in particular should you be using here?)

We should be using hardware that supports SPMD, simple program multiple data. In each thread each letter can be checked independently of each other. We can assign a different processor to check a different word in each thread, I would consider using interleaved assignment over blocked. In blocked a processor can get tied down doing an enormous calculation.

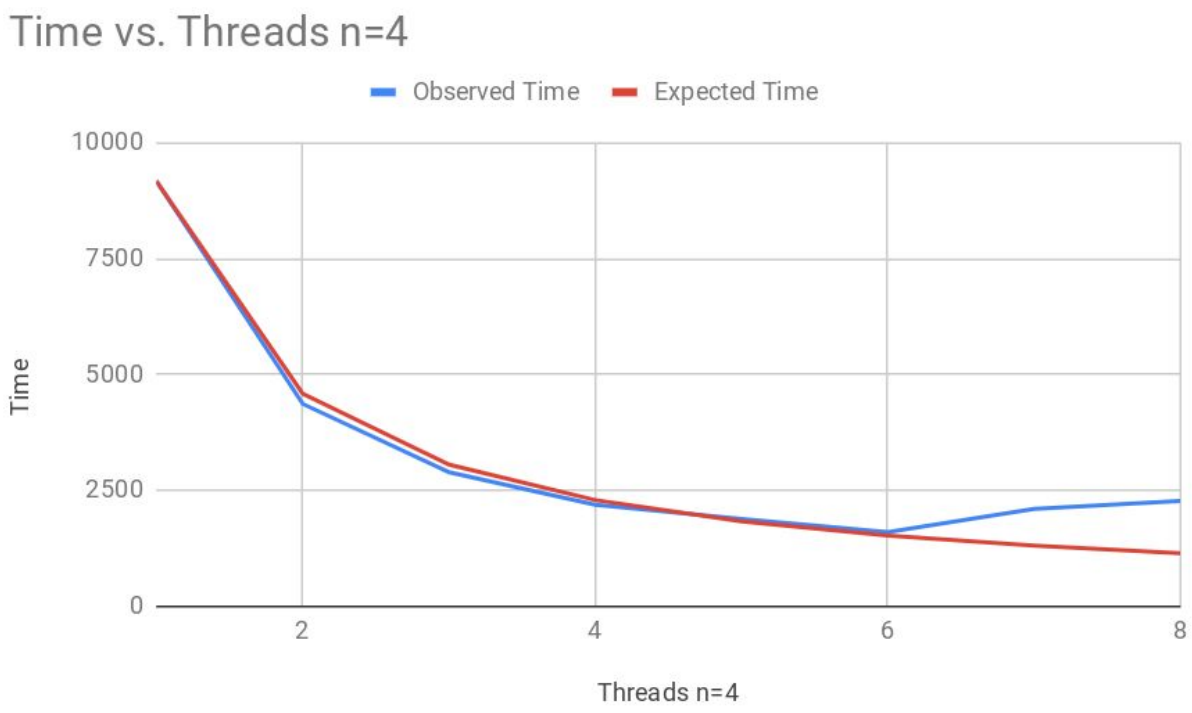
Is speedup linear in the number of cores used? In the number of threads used? Please justify why yes or why not.

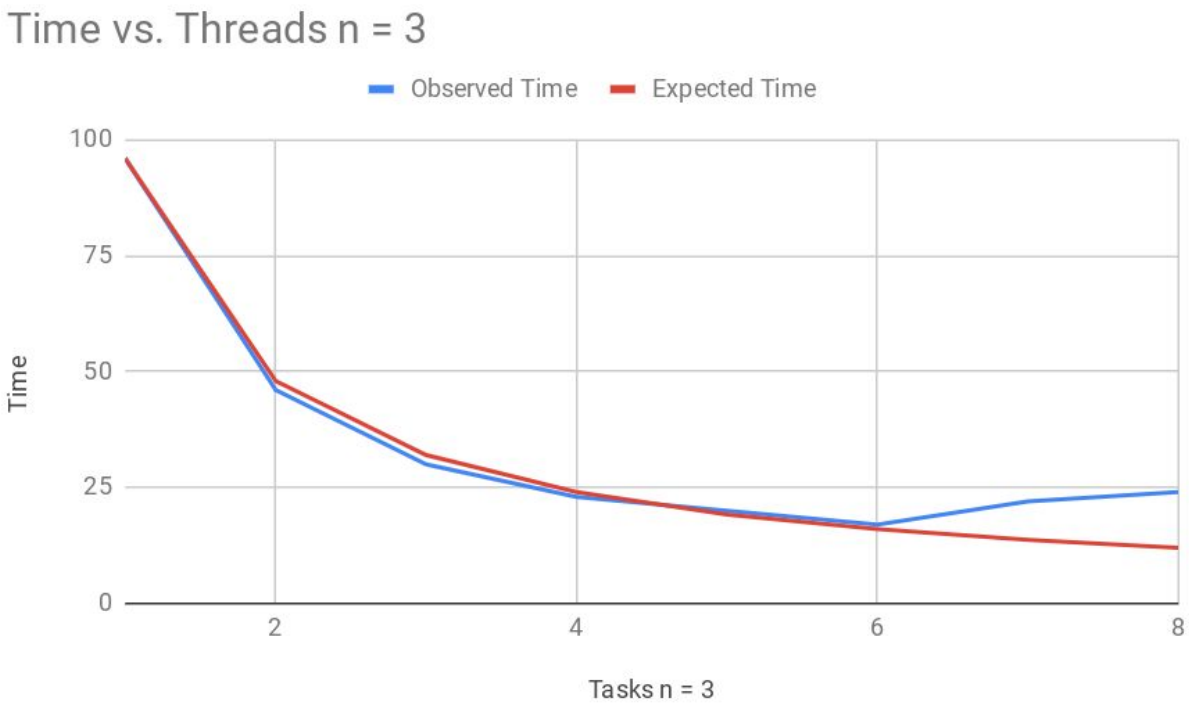
Speedup is linear based on the number of cores. Adding more cores adds more processors. This has a linear increase in time.Based on our results speedup is not linear with number of threads. As you can see from our graphs our time vs task graph is not linear. The greatest improvement happened from no tasks to 2.

Results and findings:
In this version of the password cracker we used pthreads to implement SPMD. In our design we made sure that each thread had an equal workload, and was optimized to find the password equally across all threads. We split up the workload by starting prefix. This meant that each thread would get a specific section of the total number of possible permutations for a certain length. For example, say we are comparing hashes for length n=3. Each thread would get $(3)^{\text{<num chars> / <num threads>}}$ possible permutations to work with. Because each permutation of the same length should take the same time, every thread has the same workload. We iteratively increase the length of the password we're trying to find. So we try n=1, then n=2, then n=3 etc. This is better than each thread having its own length to calculate, because that is obviously an unequal workload.
Our results showed that the method we used for splitting up the workload did lead to an almost linear increase in computation time versus the amount of cores used. Below are graphs and tables from our findings. We used the password "zzzz" for chars n=4, and "zzz" for n=3. Our system is a hexa core processor, but we used up to 8 threads to test what happens when we

use more threads than our system has cores. From the data our observed time closely matches the expected time up to where the threads>cores. After the threads>cores, we see a slowdown in performance. This is most likely due to the overhead incurred when scheduling threads to be run on cores. With more threads than cores we must split time for the same workload, but now we must context switch every so often since not all threads can fit in the execution context.

Threads n=4	Observed Time	Expected Time	Threads n = 3	Observed Time	Expected Time
1	9176	9176	1	96	96
2	4370	4588	2	46	48
3	2894	3058.666667	3	30	32
4	2191	2294	4	23	24
5	1887	1835.2	5	20	19.2
6	1604	1529.333333	6	17	16
7	2106	1310.857143	7	22	13.71428571
8	2273	1147	8	24	12




Time vs. Threads n = 3
Observed Time
Expected Time
100
75
50
25
0
Time
2
4
6
8
Tasks n = 3