Mini project report by
D Krishna Pawan - CS18BTECH11008
K Vamshi Krishna Reddy - CS18BTECH11024

What was the project about? :

- The aim of this project was to detect attacks on programmable network systems which are performed by sending unusual traffic by the network attackers.

- The idea is to first model and produce a behaviour for the network infrastructure under normal conditions. Under no attack scenario, there would not be much deviation from this captured behaviour. Now we monitor the system at runtime and compare this behaviour with our normal behaviour at periodic intervals. If there is any significant deviation, then we report there is a possibility of attack.

- The detection of attacks was simulated in the NetCache system (basically an in-network cache that can be implemented in programmable switches to achieve speed and load balancing using minimal memory in a key-value store architecture). In this architecture, usually a small set of keys are queried for a very large number of times (60-90%) and hence storing these “hot keys” in netcache would be beneficial. The paper and code repository to the NetCache system is already available. Some augmentation was made to the existing code repository to capture the paths of different packets using counters.

- Zipf distribution was assumed as the traffic for the NetCache system under normal conditions. In this distribution, we note that a small set of keys appear a large number of times.

- The attack traffic was generated in such a way that there would be rapid changes in hot items and the queries load would be taken dominantly by the servers thereby reducing the performance provided by the netcache. The attack scenario is simulated by combining both attack traffic and normal (zipf) traffic.

- We plot the graphs for load on servers and also statistical tests such as KS test or Chi-Square test which gives the deviation between observed and expected behaviour.

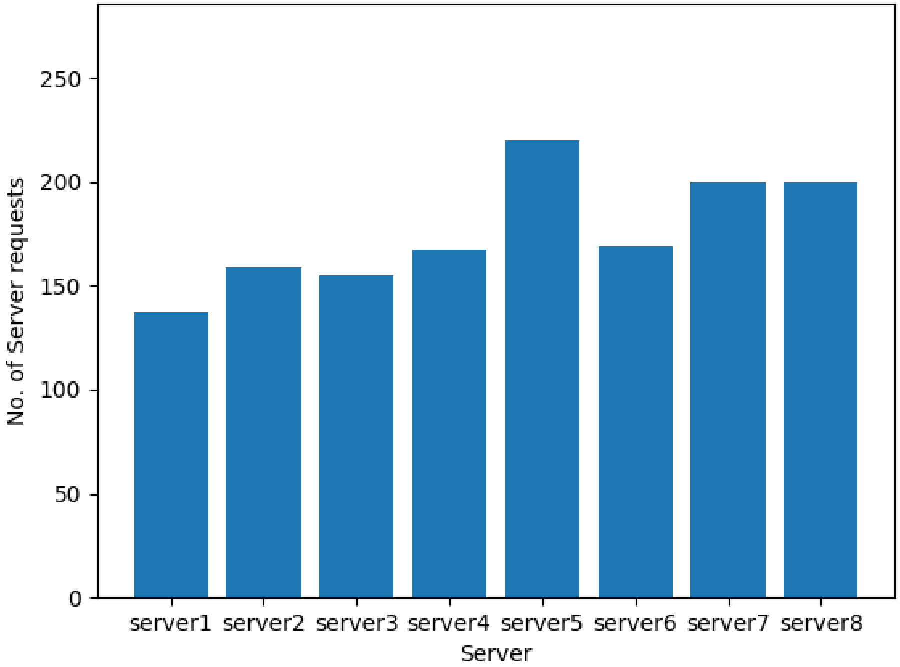
Creating an attack scenario :

In the implementation of netcache, When a particular key is queried more than k number of times(where k is the threshold for deciding whether a key is hotkey or not) in a particular interval of time, those keys are marked as hot keys (in case of netcache, k is 3). The attacker can simulate an attack by estimating this k and then querying each item k number of times thereby forcing a large number of hotread reports and cache updation which could potentially remove the popularly queried keys of the zipf traffic.

Since the real world traffic should consist of both zipf and attack traffic, both of them were merged as follows:

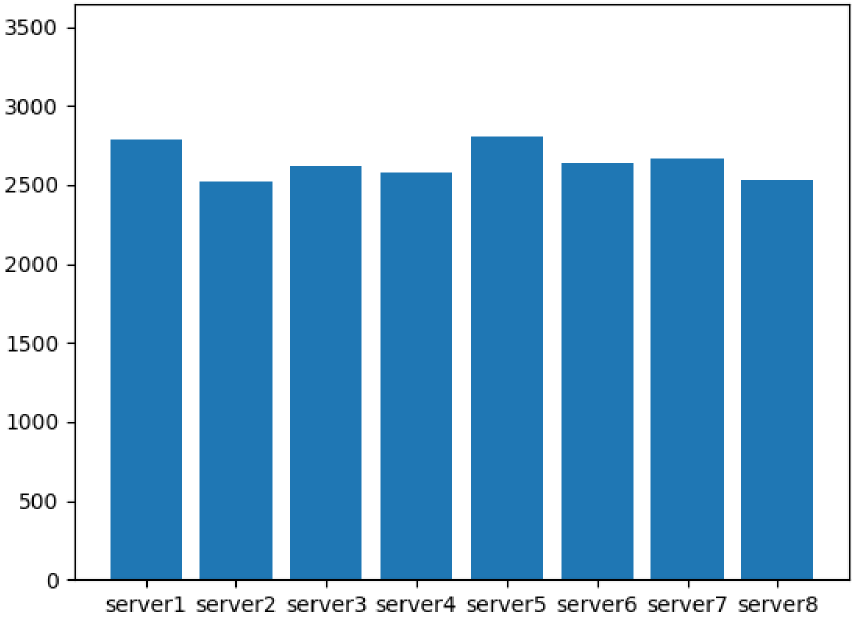
- Initial few windows consisting only zipf traffic
- Followed by small number of windows which would consist of around 80% attack traffic and 20% normal zipf traffic
- Remaining final windows consisting only zipf traffic
- attack traffic in the merged file is approximately 20% of the total traffic
- **NOTE:** attack.py generates attack.txt file with 100000 attack queries. merge_zipf_attack.py reads from files attack.txt and zipf_sample_100000_05.txt. These file names can be modified in line 3 & 4 of merge_zipf_attack.py

Plotting server requests with netcache in usual case :
In a general scenario without any attack, a total of 1000 queries are server queries and the rest are netcache queries



Plotting server requests with netcache in attack scenario :

In a scenario with 20000 attack queries, almost 17000 queries are server queries, which clearly implies that the attack is successful in that interval of time

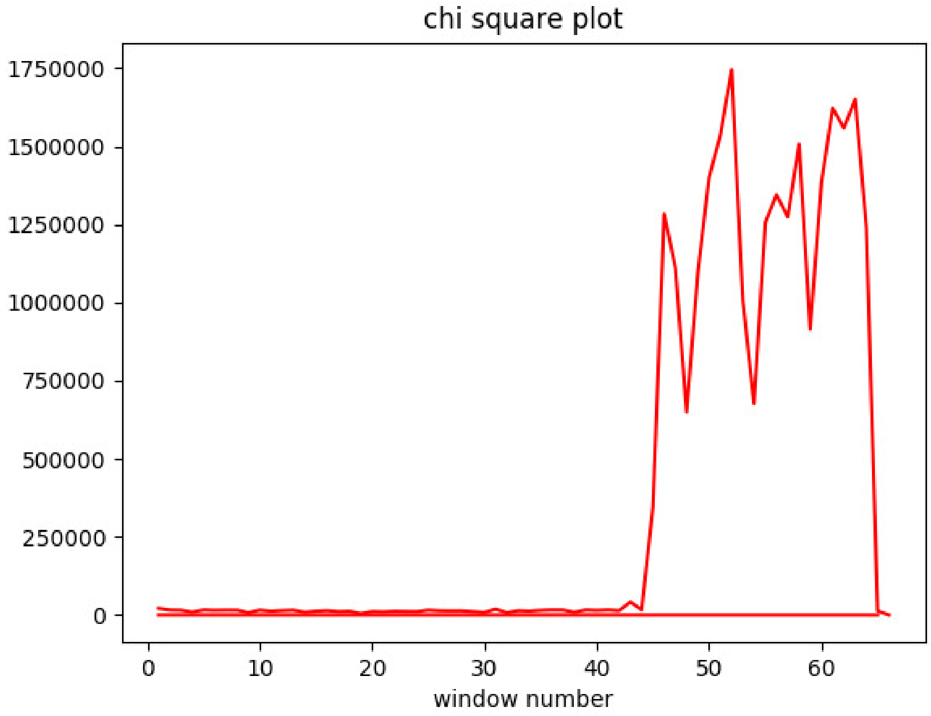


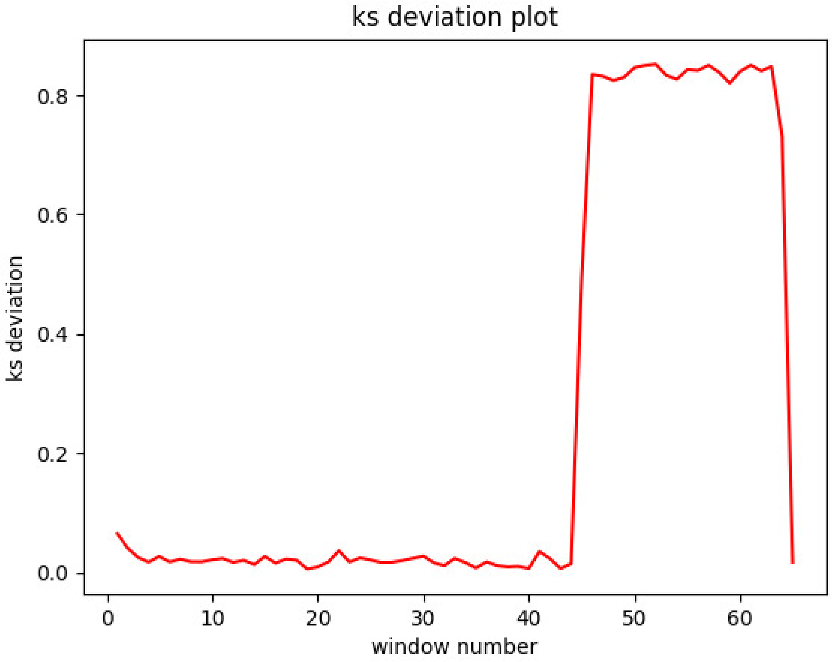
Detection of attacks :

For detection attack scenario,

- First we incorporate ball-laurus script to the p4 of the netcache design, so that we get path statistics of every packet
- Then for every 5 sec, we query the mininet to get the BL codes of paths and the respective packet count of each path and subtract the previous values to obtain the BL code and respective packet count pairs corresponding to that particular window
- Once we obtain BL codes and counts corresponding to a particular window, we can use this to :
    - First, we simulate the code with zipf distribution and obtain the output statistics of each window, which will be the expected behaviour under normal conditions
    - Next, we simulate the code with the attack distribution and obtain the output statistics of each window, which will be the observed behaviour. We compare this observed/actual values with the expected values
    - Initially Chi Square test was proposed to compare actual behaviour with expected behaviour, but a problem was that when expected value is 0, the chi square value shoots up (division by 0)
    - So, we came up with the KS test, where for both observed and expected scenarios we construct the prefix sum array of BL counts of each equivalence class and then compute absolute difference between the prefix sum arrays of observed and expected scenarios and finally return the maximum of these values as the deviation.
      This method also eliminated the above division by 0 problem in Chi Square test.

- We compared the plot deviation of ks test with chi square, which seemed to be almost similar
  Plots attached below:
  We can see the rise in deviation in attack window for both plots





Files added / modified :

- **src/kv_store/data/attack.py** - removed previous version and added new file. Generates attack traffic (attack.txt)
- **src/kv_store/data/merge_zipf_attack.py** - new file added for merging attack and zipf traffic (generates merge_zipf_attack.txt)
- **src/p4/enormal_ks.py -** new file added for using prev enormal.py and made some modifications. Generates normal.txt (has expected bl codes and counts)
- **src/p4/ewma_ks.py -** new file added for finding deviation via KS test instead of chi-square (generates ks-results.txt)
- **evaluation/README_ks.md -** made some changes as per our new files