## Executive summary:

The problem in this project is to develop a method to filter different attacks from training and testing files. The basic principle is to derive patterns to describe attack or normal packets, and then to distinguish attack packets from normal packets with these patterns.

During the test, I cannot find a perfect argument to filter all attacks, but a relative proper one. Sometimes, one argument set can capture all attack packets for one specific attack type, but same argument set works badly for another attack. Though I cannot detect all attack packets, the program still has a relative reasonable accuracy.

## Specification:

As the subsequent project of project 1, I'm using the same binary record structure for this project. In this project, there is a new file "recordMath.c" which handles all rule related functions. Three supplementary tools (printsum, split and existcheck) have been developed to assist test and development.

## Methods and techniques:

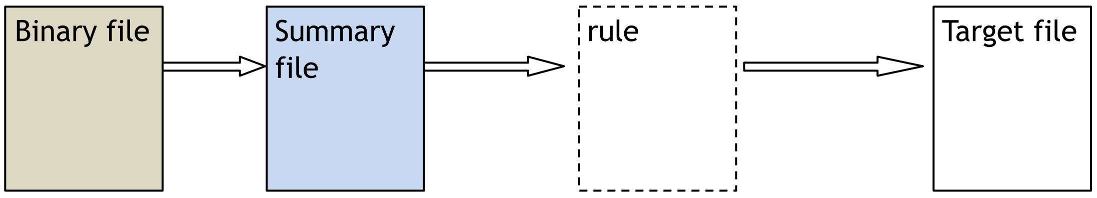
In this test, I have following assumption "value 0 does not mean anything". Because in these 41 attributes, most of them are "continue" value. If one "continue" value is 0, this is means there is no activity. So if one attribute's mean value is 0, I will not use this as the rule to filter normal/attack packets. On the contrary, the rule is based on attribute which has positive value.

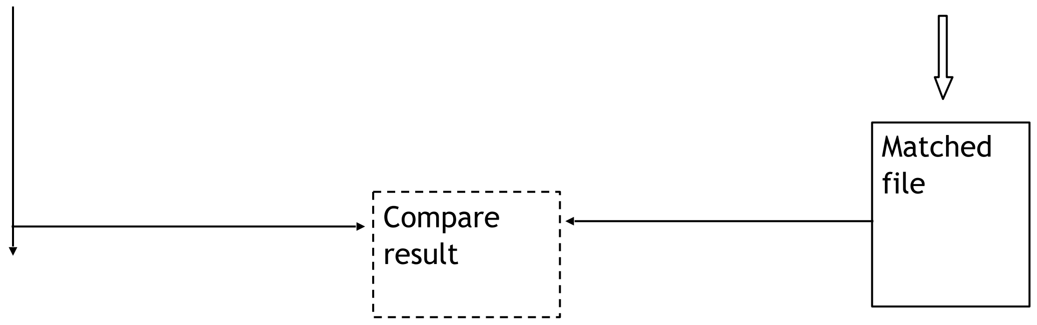
Before we go to the specific IDS (misuse and anomaly) design, there are some works in common. When there is a binary data file, the program iterates the file to have min, max and mean values of every attribute. Then the program iterates the file again. For each attribute, it counts how many entries' value is greater than the mean value (or a value which is controlled by a parameter). The reason I do this is to find how many records dominate (contribute to) the final mean value. For example, 1000 records, 950 records' values are 0, other 50 records' values are 1. The mean is 0.05. There are 50 out of 1000 are higher than the mean. So in this case, the record that has value 1 is not a common scenario, but could be an "accident". Therefore, I will not treat this attribute as a key pattern attribute. On the contrary, if there are lots of records whose values are greater than mean value or within an area of mean value, we treat this attribute as a key pattern attribute. After get the mean, min, max and overmean, the program saves these values to a file with suffix "_sum" (stands for summary).

As the input argument which specified the directory which stores these summary files, the program generates rules according to summary files. Each rule has a value range [min, max]. During the process of retrieving rule, there are two parameters. One is THRESHOLD, another is MINSHIFT. THRESHOLD decides what percentage of records whose value is greater than the mean value then we treat this attribute as a key attribute. If an attribute is not a key attribute, this means we will not use this attribute to check whether a record is normal or attack. A non-key attribute's value range is [0,1], this allows any value pass through the check. MINSHIFT controls the min value (low bound). Given an attribute is a key attribute, the low bound is set by mean*MINSHIFT. The reason does not use statistic min value for the low bound is because it introduces a high false positive for some attacks, though it can include all attacks (see Results section for detail of parameter selection). All summary files under that directory will generate a rule set. Then the program iterates the target file (training_bin or testing_bin) to check each record, to see whether its 41 attributes value are within rule's relevant attribute value range [min, max]. The pseudo code is as follows:

```
/*******************/
For each record i of target file
    For each rule j in ruleset
        For each attribute k
          If  (record[i]'s attribute_value[k] >= rule[j]'s attribute[k].min) &&
                (record[i]'s attribute_value[k] <= rule[j]'s attribute[k].max)
             {  //continue;}
         Else {// does not match}
/*******************/
```

After find a matched record for a specific attack type, this record will be saved to a new file with suffix "_matching". Program existcheck will check how many records in this file are in its original file. The rough process is as following diagram.





1) Misuse IDS design and operation:

   In this case, I select five attacks: IPSweep, Back, PortSweep, NMap, Pod. The reason I select these 5 attacks is because each of them have many (thousands) records. The program generates 5 summary files for these 5 attacks. Please see "Results" section for detail file information. The program loads these 5 files and retrieve 5 rules to construct a ruleset. With this ruleset, the program goes to check every record in the training file.

2) Anomaly IDS design and operation

   In this case, a summary file is generated for Optimized_Nomal_bin. The program loads this summary file and generates a rule. With this rule, program filters the normal packets from training file.

**Implementation:**

The main part of the program includes following functions:

1) void recordsDir(char *dirName)

   /*Calculate the mean, min max value and count overmean number from attack records. We only check the attack file with more than 20 records.*/

2) int recordExist(struct RECORD_S *rds, struct FILE_INFO *fileinfo)

   /*this function check whether the record is in specified file*/

3) void applyRulesToFile(struct FILE_INFO *fileinfo, struct RULE *ruleset)

/*this function tries to apply a rule to filter records in specified file. The rule is a "record pair" which defines the min and max value of attributes. If entry is within this rnage, we copy this entry to the matching file.*/

4) void loadRuleFromFile(struct RECORD_S *min_rds, struct RECORD_S *max_rds,

   char *fileName)

   /*this function loads the information from summary file and conclude it to a rule which stores in the min_rds and max_rds*/

5) void applyRulesToDir(char *dirName, char *targetFileName)

   /*this function pick every rule from specified dir, and apply this rule to a target file.*/

Please see source files for detail.

**Results:**

**1) misuse IDS**

In this test, I select 5 attacks: IPSweep, Back, PortSweep, NMap and Pod. With the existcheck program, for each attack type, I count how many attack packets exist in the training file. The result is as flows:

|  | IPSweep | Back | PortSweep | NMap | Pod |
|---|---|---|---|---|---|
|  | 2568 | 591 | 1962 | 1053 | 147 |

Training file has 548911 packets in total.

The intuitive thought is to use statistic min and max for the rule. But the result is very bad. For attack IPSweep, the statistic min/max rule captures 176681 packets as suspected IPSweep attack packets. But there are only 2568 IPSweep packets in the training file. Even it includes all 2568 attack packets, the false positive is (176681-2568)/176681 = 98.5%. Such high false positive rate is not allowed.

So, I do another test with THRESHOLD=0.6, MINSHIFT=0.2.

“Note A”: X/Y/Z . X: how many packets are real attack packets (exist in its original file); Y: captured suspected packets; Z: the actual attack packets exist in the training file.

False positive rate = (Y-X)/Y

False negative rate = (Z-X)/Z

| | Note A | False positive | False negative |
|---|---|---|---|
| IPSweep | 1802/4207/2568 | 57.2% | 29.8% |
| Back | 547/547/591 | 0% | 7.4% |
| PortSweep | 1780/21875/1962 | 91.8% | 9.3% |
| NMap | 662/1115/1053 | 43% | 37.1% |
| Pod | 144/144/147 | 0% | 2% |

Following log shows the existcheck result when comparing captured suspected NMap attack packets to the original NMap file (contains all NMap attack packets). The “. ” stands for exist, “X” stands for does not exist.

```
        biru@ubuntu:~/Work/ISS/project/src/existcheck_src$ ./ized_attacks_normal/test/attack2/Optimized_NMap_bin_sum_matching ~/
Work/ISS/optimized_attacks_normal/test/Optimized_NMap_bin

        matchfile=[/home/biru/Work/ISS/optimized_attacks_normal/test/attack2/Optimized_NMap_bin_sum_matching]

        originalfile=[/home/biru/Work/ISS/optimized_attacks_normal/test/Optimized_NMap_bin]

        X....X.....XXXXX..XX..XX.X...XX..X.X.X......X.XXXX.X..X..XX.X...........X.X...X...X.XX.X.XX..XX.XX.X....XX..X.X.X......X.XX.
........X.X..XXX..XX.........X...X....XXX..XX.X..XX.XXX.X.....XXX..XX.X.X..X...X.X.XXXXXX.X.XX.XX..X.....X..X.XX.XXXX.X...X.
.X.XX.X.XX.....X.XX.XX.XX.X......XX.X.XXX.X......X..XX.......XX.....XX.X......X...X.X...X.X..XX.....X.XX.X.X.XX...X.XXX...X.X...X.
...X.X..XXX.....X..X.....X.....X..X.....XX.....X..XXX...XXXXX..X..X.......XX..X...X.......X.XX.X..X..X...X.X...XXX.XXXXXXX.X...XX..X
X....X.X.X.....X...X.XXX..XXX..X..XXXXX......XX.XXXXX.X.X.X.X..X...XX.X...X.X..XX.XX.X....XXXX.XXXX.X..XXXX.X...X..X.
XX.X..XX....X.X.XXX.X.XX.XX.....X...X...X.X..X..X...XXX..XXX.X.X...X.....X....X....X.....X.XXX....X..XX.XX..X..X...XX..X..XX....X.
XXX...X.X..XXXX..X..XXX.XXX..X.X...X....X...X.XX.........X.....X.X....X..XX.XXXX..X....XX.XX.XX.XX.....X.XXX..X.XXX.XX.XX..
.....X.X.X...XX.X..X..X.....XX....X......X............XX..X.X........X.X...XXX.X..X.XX..XX..X.XX.XX..X....X.X.XXXX.X.X..XX...XXX..XX
X..X.....X...XX.X..X.....XX.X.....X.X.........X.X.X..XXX.XXX..X.X..X.X.XX..XXX.XX.XX.XXX..X..X.XX.X..X.X

 There are totally [662] of [1115] records accord to original file.

time consumes: 2 seconds
```

The reason could be a small MINSHIFT lowers the low bound which allows many normal packets in. So I go back to adjust the parameter, set THRESHOLD=0.6, MINSHIFT=1 (MINSHIFT=1 stands for using mean*1 as the low bound). Rule range is [mean, max]. But in this case, it will definitely miss some attack packets.

Result is :

|  | Note A | False positive | False negative |
|---|---|---|---|
| IPSweep | 1802/2464/2568 | 26% | 29.8% |
| Back | 536/536/591 | 0% | 9% |
| PortSweep | 830/830/1962 | 0% | 57.7% |
| NMap | 657/671/1053 | 2% | 36.3% |
| Pod | 144/144/147 | 0% | 2% |

So it looks MINSHIFT=1 is proper. Then I go to adjust THRESHOLD. The THRESHOLD decides what percentage of records contribute to the mean is treated as a key attribute. If THRESHOLD is low, this means for an attribute, even with the case that the mean is dominated by a small group, we still treat this attribute as a key attribute.  The there are more key attributes. This means the check is stricter. As a consequence, since MINSHIFT=1, the low bound is mean value, only small number of entries can pass the filter.  If THRESHOLD is high, this means it focus on "exact" key attribute that the mean value is dominated by major entries. But the consequence is it will have small number key attributes to construct the filters. Then more packets will be captured. See following result which I use Optimized_IPSweep as an example.

|  | THRESHOLD=0.2,MINSHIFT=1 | THRESHOLD=0.9,MINSHIFT=1 |
|---|---|---|
| Captured | 0 | 389203 |
| Actual IPSweep attack packet in traning file. | 2568 | 2568 |
|  |  |  |

The result shows neither THRESHOLD=0.2 nor THRESHOLD=0.9 works well.

The resource cost of the IDS check:

1) CPU: since the system is pre-emption , CPU will be used on program if there is no other program running.

2) Memory: there is no dynamic allocated memory blocks; static memory is limited.

3) time:  it costs at most 10 seconds. Since all rules are in memory (just a min-max pair), and it iterates target file once. The real time consuming part is existcheck. When there are N packets captured, and its original file has M packets. Then checking how many packets of these N packets are in the original M packets file will costs N*M times comparison. Because file is too big, it cannot be loaded into memory, every comparison introduces a file reading. We will discuss this in following anomaly IDS section.

**2) anomaly IDS**

As the test in the misuse IDS, I have a relative proper argument pair (THRESHOLD=0.6, MINSHIFT=1). So I directly use these two parameters for anomaly IDS because the behind algorithm is same. The only difference is anomaly IDS retrieve the rule from normal packets.

In the training_bin file, there are totally 548911 packets, after applied the rule which derived from Optimized_Normal_bin_sum, I captured 286245 packets from the training file. At first, I run the existcheck program directly to check how many packets in captured file (286245) is in the original Optimized_Normal_bin (which has 576710 packets). But the program keeps on running and does not exit, even after I go home and get back to school for a night. Then I write a program split to split the Optimized_Normal_bin_sum_matching (captured file) to many small files, each file contains 4000 records.

Following log shows the test result of running existcheck for one small file Optimized_Normal_bin_sum_matching0 .It costs 709 seconds. Every "." stands for a match. Every "X" stands for a dis-match (cannot find this packet in original Optimized_Normal_bin file).

```
biru@ubuntu:~/Work/ISS/project/src/existcheck_src$ ./existcheck ~/Work/ISS/optimized_attacks_normal/test/attack/
Optimized_Normal_bin_sum_matching0 ~/Work/ISS/optimized_attacks_normal/test/Optimized_Normal_bin

matchfile=[/home/biru/Work/ISS/optimized_attacks_normal/test/attack/Optimized_Normal_bin_sum_matching0]

originalfile=[/home/biru/Work/ISS/optimized_attacks_normal/test/Optimized_Normal_bin]

....................................................................................................................................................................................................................................
....................................................................................................................................................................................................................................
....................................................................................................................................................................................................................................
....................................................................................................................................................................................................................................
..............X....................................................................................................................................................................................................................
...................................X............................................................................................................................................................................................
....................................................................................................................................................................................................................................
..............................................................................................................................................X.....................................................................................
..........................................................................................................................................................X.........................................................................
....................................................................................................................................................................................................................................
```

```
..............................................................................................................................................................................................................................
..............................................................................................................................................................................................................................
..............................................................................................................................................................................................................................
..............................................................................................................................................................................................................................
..............................................................................................................................................................................................................................
..............................................................................................................................................................................................................................
..............X...............................................................................................................................................................................................................
.................................

 There are totally [3995] of [4000] records accord to original file.

time consuming: 709 seconds
```

If I run all these files, given they all consume 709 seconds, there are totally 71 files. It costs 709*71 = 50339 seconds = 13.98 hours (that's why the program does not complete even after running for a night.)  So I select several files to run the existcheck.  The result is as follows:

| | Time (seconds) | | False positive | |
|---|---|---|---|---|
| File 0 | 709 | 3995/4000 | 0.125% | |
| File 17 | 1038 | 3996/4000 | 0.1% | |
| File 29 | 893 | 3996/4000 | 0.1% | |
| File 53 | 737 | 3994/4000 | 0.15% | |

Since I do not run the test for all files, the false negative rate is estimated as follows.

There are 286245 packets captured, and 384229 normal packets in the training file. Given the false positive rate is 0.15%. Then there are 286245*(1-0.15%) = 285816 packets are actual normal.  The false negative rate is (384229-285816)/384229 = 25.6%.

In this project, I only use training file for the test is because the training file and testing file are same to my IDS. The model (argument adjustment) is based on original files (separate attack and normal files).

**Further thinking:**

1) how to increase the file record comparison speed:  the slow comparison greatly slow the development and test speed. Create a hash for every record could be a solution. Stores the hash value as the 42 attribute value for each record. Every time load the file, first load these hash values and construct a table in memory. But if the file is very

big, even this memory hash table will be very big. I did not have time to try this in this project.

2) At first, I'm thinking of using existed distribution model to evaluate the possibility of the attribute value, such as the standard deviation. But these values are not standard normal distribution. Then I turn to the number of packets dominate the mean value. If we can just have limited number of parameters (the less, the better), and have a rule for every parameter (like what's the result if change this parameter lower or higher), then maybe we can have a dynamic system which can adjust the parameter according to real-time calculated accurate rate.

**Source files:**

Under the src/ there are three directories existcheck/ split/ and printsum/. These three directories are for three small testing tools. Src/ contains main program's source file.