Mu Lin

August 2, 2016

Bipartite Match Algorithm in Byte Stream Inspection

The conventional way to inspection byte streams, in particular web traffic, often uses automata (DFA/NFA) or regex, it works well when the regex is relatively "simple", when evasive techniques are used, automata is less effective.

This article propose a new way to inspect byte streams using bipartite match algorithm, effectively, the token is kept simple, the complexity of the composition of the pattern (formed using token) can be expressed in bipartite match terminology.

Pattern, which is a sequence of finite number of plain vanilla signatures (regex and others) with modifiers, such as
P = {sig1,sig2,sig3}(mod1, mod2) /* not exact syntax */
Note a token can match to multiple sigs, such as string "abc" can match to sig1 and sig2.

The nodes in the sequence are adjacent and their order of appearance is significant, the same as the plain vanilla signatures come to be expected, these two implicit are inconvenient for describing collections of signatures where order or continuity or both are not the matching criteria.
Therefore, two modifiers are introduced:
mod1 = order-free
mod2 = continuity-free

There is another modifier called random which is to describe the exact position where the continuity of the sequence stops, such as
/* between sig1 and sig2 there is token whose signature we don't care */
P = {sig1, sig2, sig3}(random1)

random is a special case of the more general continuity-free modifier, we only discuss continuity-free here since as long as we can handle continuity-free, we must be able to handle random modifier.
The above modifiers are expensive to execute, if not possible in DFA, compound with signature polymorphism which calls for NFA, the matter is even worse.

Ab Initio, from the first principle, this is combinatorial pattern matching problem.

We hereby describe a two stage dynamic programming algorithm and related data structure as an alternative to dfa.

This algorithm matches signatures once and only once just like dfa in the first stage, and uses polynomial runtime to select matching pattern in the second stage.

Conceptually,

1. In compile time, the patterns are compiled into a n*m binary matrix plus one additional column to describe the modifiers, where n is number of patterns, m is number of unique signatures used in the

patterns. The signatures are compiled into dfa as usual(partitioned per url host+path using a trie) such that
e(i,j)=1 if and only if ith pattern has jth signature. (2)

2. In the first stage of runtime, we first build a n*m binary matrix where n is number of tokens matched against the dfa(tokens are obtained from protocol/url parsing phase), m is the number of signatures such that
e(i,j)=true if and only if ith token matches jth signature, note the ith row can have multiple columns set to true.

3. In the second stage of runtime, we obtains the sub runtime n*m matrix for a particular pattern such that
e(i,j)=true if and only if ith token matches jth signatures in the pattern.
4. bipartite graph is equivalent to matrix, therefore, we can build a bipartite graph where there are two sets of vertex (n & m, n is token, m is signature), there is a weight 1 edge between ith vertex in n and nth vertex in m if e(i,j) in the matrix is true.

We first claim that the necessary condition for a pattern match is the bipartite graph's maximum bi-partite matching or max flow equals to m, the number of signatures in the pattern.

We can prove the above using definition of maximum bipartite matching/max flow, let us call it Raven Pattern Matching Theorem L.

We further claim and can prove that Theorem L is not only the necessary but also the sufficient condition for patterns who have both order-free and continuity-free modifiers.

When patterns have only order-free modifier, we need to additionally check it is continuous max flow. Ford-Fulkerson algorithm (FFA) and its variants can determine the max flow number in O(Ef).

```
Function FFA
        for each edge (u,v) in E(G)
                do f[u, v] = 0
                f[v, u] = 0
        while there is a path p from s to t in the residual network Gf
                do m = min{c(u, v)-f[u, v]: (u, v) is on p}
                for each edge (u, v) on p
                        do f[u, v] = f[u, v] + m
                        f[v, u] = - f[u, v]
```

FFA in fact is overkill since it deals with general graphs, Hopcroft-Karp algorithm (HKA) is the max flow algorithm for bipartite graph. It runs in O(E*sqrt(V)), and for random graphs, it runs in near linear time, for sparse graph, HKA is best in worst case performance.

```
Function HKA
        for each u in U
                Pair_U[u] = NIL
        for each v in V
                Pair_V[v] = NIL
        matching = 0
        while BFS() == true
                for each u in U
                        if Pair_U[u] == NIL
                                if DFS(u) == true
                                        matching = matching + 1
```

return matching

We can prove if and only if the diagonal of the sub square m*m matrix are all true then it matches a pattern who has neither order-free nor continuity-free.

A naive algorithm can determine the above in O(nm), Knuth-Morris-Pratt style algorithm can be used to speed it up.

Finally we can prove that the runtime is O(n) for patterns has only continuity-free.

All 4 combinations of order-free and continuity-free are therefore having polynomial runtime complexity or better.
random sigs modifier is quite easy to implement in bipartite graph, e(i,j)=true for all i, where j is random sig's column, Do Not Care (X) as in truth table is a better name.

repeater is also easy to implement in bipartite graph, e(i,j)=e(i,j-1) for all i, where j is repeater's column.

group can be supported, its behavior with modifiers are yet to be determined.
In conclusion, just like dfa is the mathematic model for regex, bipartite graph is a good mathematic model to represent pattern.

combinatorial analysis, the problem space is roughly P(k, n) * C(k, m) for k = 1 .. n
(1) O(Ef) where E is the number of edges, f is max flow.
(2) when order-free and continuity-free are both specified in the pattern, it is better to specify the max number of params in order to have the algorithm being bounded, the same for matching the signatures in the first stage.