Packet Matching Algorithms

Abstract
In order to identify a threat, your intrusion prevention system will match incoming packets and flows against a database of pre-defined features. Organizational networks will however run at Gigabit (if not tens of Gigabit) speed, thus these matching routines will need to be sufficiently optimized to accomplish full coverage of the monitored network traffic.
In this part, you will experiment with the iptables firewall included in the Linux operating system and measure its performance to increasing rule sets. Based on our observation, we will review high-performance algorithms to match simple entries such as port numbers or flag combinations as well as introduce data structures to efficiently compare and match sets, an operation we for example need to test IP addresses.

Your organization is under a denial-of-service attack: An unknown adversary is exploiting a large amount of open DNS resolvers to flood the uplink of your company, and as a result the services you host in your DMZ are no longer available to the outside. Given the normal network traffic and requirements of your organization, it is not possible to simply filter out DNS packets, but your administrator team has identified that the reflective attack originates from approximately 2 million sources. Your organization happens to have devices with packet filtering capability installed where your Internet uplink terminates with your ISP. The plan of action is to implement the appropriate rules to filter out the traffic at the well-provisioned edge and thus stop the overflow of your capacity-limited uplink.

From this use case we see that building an intrusion prevention system also involves the development of efficient algorithms for packet handling and analysis. As a packet has to be matched against dozens if not hundreds of rules to see whether it is malicious or not, and to determine whether it should be dropped or may be permitted into the network, this lookup procedure must be designed as efficient as possible to not delay the packet longer than necessary and reduce the computational workload for the device carrying out the scanning. In this part of the project we will look a bit deeper into data structures and algorithms within the context of an IPS's packet filtering.

As a starting point for the analysis, we will first benchmark the packet filter performance included in the Linux operating system, which was discussed in a case study in chapter 4. To investigate which algorithms iptables uses internally and its packet parsing performance, you will build an application that will create and inject packets from a set of random origins, which will be parsed by the iptables packet filter and by using an appropriate post-processing rule will be delivered again to your test application.

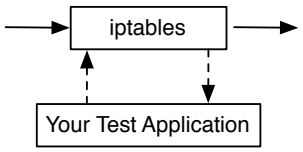

Figure 1

Recognizing and matching a previously injected packet against the frame returned on the other side of the firewall, your test application can compute the average delay and standard deviation of the packet processing within iptables. There are multiple configurations to achieve this, the simplest one being an application with a sending and a listening socket whose traffic is redirected in the OUTPUT filter and selectively dropped in the INPUT filter. In order to quantify the delay iptables introduces and its dependency on the amount of rules in the table, you will run your test application iteratively with a ruleset of different length in each run. If you would try to filter out the attack traffic from the known DNS resolvers, these rules would naturally come first in the filter chain to drop a packet with an IP address match, before general rules or the chain default policy will apply.

> **Task 1** *Measure the one-way latency of packets from your test application between zero and 2 million firewall rules in increments of 100,000. Show the average delay and standard deviation in a graph, what can you conclude about the algorithm used internally for packet matching by iptables?* ∎

The results highlight that the design of a firewall configuration must not only be carefully planned based on the security design, trust zone compartmentalization and maintainability of a network, but also in terms of how filtering rules are loaded into the device. Investigate two alternative options to your original problem, DDoS mitigation by filtering attack traffic from a list of known IP sources, based on the documentation of iptables as well as forum posts and mailing lists where a person faced a similar problem.

> **Task 2** *Describe a method (a) to structure the same filtering rule set differently to increase the performance, and (b) to make the internal pattern matching more efficiently, for example based on ipsets. Measure the performance characteristics again for each configuration change.* ∎

In the following, we will discuss algorithms for packet header matching that can increase the performance in an IPS system. In a practical environment and at the network layer, administrators care to identify packets that match one predefined property, or whenever multiple fields in a packet assume specific values. There are algorithms well suited for each of the specific use cases, some of which you will learn about and implement below.

Single Field Matching

The most basic type of matching operation inside a firewall or intrusion detection system involves the lookup of a single attribute or property of a particular packet against a pre-defined list. An IP address may be compared against a list of specifically allowed or denied sources, or a decision how to proceed further is made on a select header field such as the IP identification number or TCP port number. The most basic but also most wasteful technique is an exact matching based on direct addressing: in a list of 2^{16} possible TCP port numbers as a lookup key to find the desired response, a single lookup allows the device to make the decision whether to forward or discard the packet. If the number of items stored in the lookup table is significantly less than the total possible key space $[0,...,n-1]$, hash tables provide a better trade-off in terms of space vs. lookup time. Instead of keeping a mapping of all possible keys to their corresponding response, the element x is hashed to a smaller space $[0,...m-1]$, and the key-value pair placed at the index of the hash $h(x)$.

In many packet processing applications, the requirements to the data structure are however much simpler. Often, it is sufficient to only know whether an element is actually included in a set, not its position and neither its associated value. While we could enumerate all IP addresses together with a bit indicating an accept or drop, it is much easier to only store all denied (or allowed – depending on the size of the filtering rule) IP addresses in a set. If an incoming packet matches an element in the set, the frame is dropped otherwise forwarded. This significantly slims down the required memory space.

Bloom Filters

An example of such a data structure are Bloom filters, which keep as the only memory a bit-vector of length m. Each element x is mapped to a subset of these m bits, based on the set of k hash functions. Each hash function h_i results in an entry between 0 and $m-1$. If an element is added to the set, the corresponding bit $h_i(x)$ in the bit vectors are set to 1 for all k hashes. To check whether an element is included in the set, the k hash functions are computed for the element x. If all bit positions $h_i(x)$ are equal to 1 in the vector, the element is probably contained the set, if at least one bit position is equal to 0 the element is not contained in the set.

It is important to note that a Bloom filter is only able to indicate a possible but not a definite match. Since the space of input values was compressed into a much smaller space, there is the risk of multiple collisions in the bit vector. There might be a set of input values $\{x_{ci}\}$ that results in the same outcomes for the k hash functions as the input x. Thus, an entry might be mistakenly identified as being part of the set (false positive), but a missing entry will never be returned as contained in the set. This property, the space efficiency and fast computation makes Bloom filters interesting for applications such as intrusion detection where the cost of foregoing a detection greatly outweighs the penalty of mistakenly labeling benign traffic as malicious.

It is straightforward to compute the probability of a false positive as a function of the available bit vector length. Assume a perfect hash function which with equal probability will return a bit between 0 and $m-1$. The probability for a random bit not being set is $1-\frac{1}{m}$, for e entries processed by k hash functions $(1-\frac{1}{m})^{ek}$. Thus, the probability that not all k bits are set given a Bloom filter with e entries, becomes $(1-(1-\frac{1}{m})^{ek})^k$. The false positive likelihood can hence be reduced by increasing the length of the bit vector m, or for a given m by increasing the number of hash functions.

While the performance of a Bloom filter is entirely driven by the performance of the hash functions, we can forego the application of a cryptographic hash function, as long as the output is sufficiently random. In practice, simple hash functions specifically designed for fast computation are applied within Bloom filters, for example xxhash.

> **Task 3** *Implement a Bloom filter with m = 175 million and k = 30, and as a comparison a hash table lookup and the data structure you have empirically determined for iptables. Based on your randomly generated list of 2 million IP addresses that should be blocked, measure the average lookup time from 0 to 2 million entries in increments of 100,000, and plot the results in a graph.* ∎

Longest Prefix Match

One problem specific to IP addresses is the longest prefix match. Recall from our discussion of the network layer the practice of assigning IP addresses through class-less inter-domain routing (CIDR). A /24 network thus has a network prefix length 24 bits, and reserves 8 bits (or 256 addresses) for the hosts within this network. As IP addresses can be aggregated in larger and larger blocks, IP filtering rules need to be matched to the most specific filter, as it might be desirable to apply different actions to a general group of hosts than to specific blocks or individual IP addresses.

Table 1 visualizes the problem of longest prefix matching for 8-bit addresses. As the prefix * matches every possible address, the default policy of the filter set is set to blacklisting. If an address begins with the sequence 100011, this longest prefix overrides the action of the * rule and causes the packet to be dropped instead. In longest prefix match, rules may be arbitrarily nested and interleaved. Although all host addresses

Table 1. Example of longest prefix match for an 8 bit address

*	accept
001*	drop
0011*	accept
10011*	drop
00110000	drop

beginning with 001 are dropped, any packet from an address beginning with 0011 is permitted, except the specific 8-bit address 00110000. This means that a list of prefixes has to be completely scanned until an exact match has been found, and otherwise retaining the longest prefix that matched the address until this point. If a longest prefix match list is search linear, this hence implies an O(N) search complexity.

Longest prefix matching can however greatly be sped up by rearranging the addresses into a more beneficial data structure. The standard procedure of IP address matches is by means of a trie – derived from the word *retrieval* –, which is essentially a search tree with the small modification that the value of each key is not stored in a particular tree element but through its location in the trie.

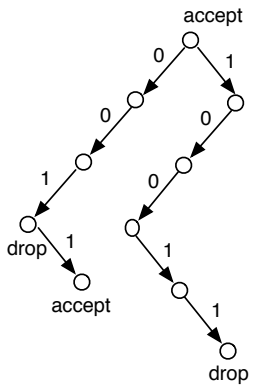

Figure 2

Figure 2 represents the four prefixes from table 1 in the format of a trie. Depending on the bit in the first address, the trie is traversed to the left or right, in case of a 0 to the left. The sequence of two 0 followed by a 1 will lead to a drop decision, unless the 1 is followed directly by another 1. From this example we immediately see that this particular trie is not the most efficient way of storing and querying the prefix list. After the initial 1 bit, four rounds of comparisons are needed to arrive at the bottom right entry 10011 and terminate with a drop, while all other branches in this part of the trie would have resulted in an accept. A radix trie or Patricia trie optimizes the basic trie in terms of space, and compacts every element with only one child into its parent. In the right hand branch, the next comparison could directly check for the bit sequence 10011 to identify the last element, while in the left hand side the comparison for successive 0s and a 1 is replaced by a direct comparison for 001.

While in principle a trie and Patricia trie has a lookup time of O(L) for L-bit prefix lengths, the specializations of tries can further improve upon this. Instead of making a single binary decision at each node, a k-stride multi bit trie performs multiple comparisons at each node, for example a bit sequence of 00, 01, 10, and 11 will take the lookup into different parts of the trie. For a branch count of k, this setup significantly reduces the necessary exploration depth of the trie traversal and for a k-stride multi-bit trie results in a complexity of $O(\frac{L}{k})$.

It is evident that the performance of the trie depends on the degree of fragmentation of the prefix matching set. To see the difference, compare the worst case scenario of completely randomly distributed source IP addresses with source IP addresses that are concentrated in subnets, after all a group of open DNS resolvers belonging to one organization that is exploited in the attack would likely be located in the same subnet, and possibly even at adjacent IP addresses. To arrive at a more realistic – yet still very simplistic – evaluation of trie performance, obtain the list of current IP address allocations from http://thyme.apnic.net/current/data-raw-table. Assume that hosts from each autonomous system (AS) are equally taking part in the DDoS attack, proportional to the total amount of IP addresses assigned to that AS. Instead of being randomly placed, the attacking IP addresses are located at a random part of the allocation but all addresses are consecutive. Implement a regular trie as well as a 2-stride multi bit trie, and for a test set of random IP addresses test how many lookups each data structure would require.

> **Task 4** *Compare the lookup performance of the trie versus a 2-stride multi bit trie under these two scenarios, a purely random origin and when IP addresses are adjacent in each subnet.* ∎