Space-efficient Structures for Detecting Port Scans

Ali Şaman Tosun*

Department of Computer Science
University of Texas at San Antonio
tosun@cs.utsa.edu

Abstract. Port scans aim to detect the services running on a computer to find vulnerabilities of a computer. Although detecting port scans using a database system is possible, it requires too much space and computational overhead and is not feasible under high load. In this paper, we propose space-efficient structures to detect parameterized versions of port scans. We investigate both exact and approximate structures for the problems. Proposed schemes are lightweight, require low space overhead, low computational overhead and can handle high load.

1 Introduction

Port scanning is the process of connecting to TCP and UDP ports on the target system to determine what services are running or are in a listening state. Identifying listening ports is critical to determining the services running, and consequently the vulnerabilities present. Many protocols and applications have reserved port numbers. For example, http protocol uses port 80, Microsoft SQL Server uses port 1433 and MySQL uses port 3306. Each machine has $2^{16}$ ports and by scanning through all the ports and testing whether any applications are running on these ports reveals a lot about the system. Many port scanning utilities are available. Utilities for Unix/Linux include strobe, netcat and nmap. Utilities for windows include SuperScan, WinScan and ipEye. Given the number of tools available for port scanning, detecting port scans is important. A connection between two machines can be identified using 5-tuple ($IP_{src}, Port_{src}, IP_{dest}, Port_{dest}, Time$) where $IP_{src}$ is the IP address of the machine initiating the connection, $Port_{src}$ is the port number used for the connection by the initiating machine, $IP_{dest}$ is the IP address of the destination machine, $Port_{dest}$ is the port number for the destination machine and $Time$ is the timestamp of the connection. This 5-tuple can be used to detect portscans.

We address the following problems to detect portscans in this paper. We represent the problems with parameters that can be set by the system administrators.

- PORT-ALERT: alert when there are connection attempts to $\geq \tau$ ports of a machine

* Partially supported by Center for Infrastructure Assurance and Security at UTSA
PORT-NODE-ALERT: alert when there are connection attempts to \( \geq \tau \) ports of a machine from \( \leq \kappa \) nodes.

PORT-NODE-TIME-ALERT: alert when there are connection attempts to \( \geq \tau \) ports of a machine from \( \leq \kappa \) nodes in \( \leq \theta \) time.

PORT-ALERT problem only counts if at least \( \tau \) ports were used in connection attempts. It does not distinguish the number of machines used in connection attempts. PORT-NODE-ALERT problem also counts the number of distinct machines used in connection attempts. Although distributed port scans are possible, too many connection attempts from fewer nodes indicate a port scan. PORT-NODE-ALERT problem does not include the time dimension. Connection attempts in a short period indicate a port scan. So, we include the time threshold \( \theta \) in problem PORT-NODE-TIME-ALERT.

Port scans can be detected by inserting all the relevant information into a database and by querying the database. However, this requires too much space and computational overhead and is not feasible under high load. Using the 5-tuples \( (IP_{src}, Port_{src}, IP_{dest}, Port_{dest}, Time) \) approach for connections requires continuous updates to the database, continuous queries and more than 10 bytes storage per tuple.

Port scan detection can be implemented at the entry point of the network and needs to handle all the connections to all the machines on the network. To have a practical system the following properties are desirable.

- **Low space requirement:** Since too many connections are handled, space requirement per connection should be low. Attacker can send lots of data filling the data structures. So, data structures with constant space requirement or sublinear space requirement such as \( O(\log n) \) or \( O(\log \log n) \) are desirable.

- **Low computational overhead:** The data structure needs to be updated real-time. To handle all the connections computational overhead per connection should be low.

## 2 Related Work

In this section, we discuss related work on bitmap indexes, bloom filters, space-efficient data structures and port detection.

Bitmap indexes were introduced in [20]. Several bitmap encoding schemes have been developed, such as equality [20], range [8], interval [9], and workload and attribute distribution oriented [15]. Numerous performance evaluations and improvements have been performed over bitmaps [8, 23, 25, 27]. While the fast bitwise operations afforded by bitmaps are perhaps their biggest advantage, a limitation of bitmaps is the index size.

Our solution is inspired by Bloom Filters [6]. Bloom Filters are used in many applications in databases and networking including query processing [17–19], IP traceback [21, 22], per-flow measurements [10, 16], web caching [12, 13] and loop detection [24]. A survey of Bloom Filter (BF) applications is described in [7]. A BF computes \( k \) distinct independent uniform hash functions. Each
hash function returns an \( m \)-bit result and this result is used as an index into a \( 2^m \)-sized bit array. The array is initially set to zeros and bits are set as data items are inserted. Insertion of a data object is accomplished by computing the \( k \) hash function results and setting the corresponding bits to 1 in the BF. Retrieval can be performed by computing the \( k \) digests on the data in question and checking the indicated bit positions. If any of them is zero, the data item is not a member of the data set (since member data items would set the bits). If all the checked bits are set, the data item is stored in the data set with high probability. It is possible to have all the bits set by some other insertions. This is called a false positive, i.e., BF returns a result indicating the data item is in the filter but actually it is not a member of the data set. On the other hand, BF's do not cause false negatives. It is not possible to return a result that reports a member item as a non-member, i.e., member data items are always in the filter. Operation of a BF is given in Figure 1.

Exact solution to find the number of distinct elements in \( n \) items require \( \Omega(n) \) space. Approximate data structures that estimate the number of distinct elements are proposed to reduce the space complexity. Counting sketches [14] and extensions [1, 5] estimates the count in one pass using small amount of storage. FM sketches [14] use \( O(\log n) \) space to approximate the count. Using linear hash functions sketch size can be reduced to \( O(\log \log n) \)[11].

3 System Model

Each machine on the Internet has a 32 bit IP address typically represented as \( A.B.C.D \). Each letter corresponds to decimal representation of 8 bits. For example, 129.115.28.66 is the IP address of a machine in CS department. Each machine has \( 2^{16} \) ports. A connection between two machines includes port numbers as well. For example, a connection from machine \( IP_1 \) and port number \( p_1 \) to machine \( IP_2 \) and port number \( p_2 \). Each process that requires communication has a port number that it can use for communication. Port numbers
are used to demultiplex the packets received by the machine. To make routing easier IP addresses are assigned in blocks. For example, 256 IP addresses represented by 129.115.28.0 are assigned to CS department. A block of IP addresses is called a subnet and represented as \(A.B.C.D/X\). \(X\) denotes the number of bits common to all the machines in the subnet. Above block can be represented as 129.115.28.0/24. In this paper, we develop a common data structure to store all the information for a subnet. We use the notation \(n\) to denote the number of machines in the subnet. For the subnet 129.115.28.0/24, the value of \(n\) is 256.

![Fig. 2. Bitmap Approach for PORT-ALERT](image)

### 4 Proposed Schemes

In this section, we discuss how to detect each time of alert using both exact and approximate techniques.

#### 4.1 PORT-ALERT

PORT-ALERT Problem can be represented using a bitmap. Let \(n\) denotes the number of machines in the subnet. A \(n \times 2^{16}\) bitmap can be used to store connection attempts to the machines in the subnet. Block diagram of the approach is given in figure 2. When a connection attempt to machine \(A\) on port \(p\) is received, the corresponding entry in the bitmap is set to indicate this operation. In addition to the bitmap we use an array of counters to store how many ports are used. When a counter reaches the threshold \(\tau\), an alert is raised. Bitmap approach is exact since it stores all the connection attempts to ports.

Bitmap approach requires \(n2^{16}\) bits for the bitmap and \(16n\) bits for the counters. Since there are \(2^{16}\) ports, 16 bits are needed for each counter. For a 256 node subnet space requirement is about 2 megabytes which is manageable. During normal operation on the network, most of the bits will be 0. To reduce the space requirement bitmap compression schemes can be used. Several compression techniques have been developed in order to reduce bitmap size and at the same
Algorithm 1 PORT-ALERT Insertion Algorithm for connection to \((IP, p)\)

1: \(index = GetIndex(IP)\)
2: if \(bitmap(index, p) == 0\) then
3: \(bitmap(index, p) \leftarrow 1\)
4: \(portcount[index] \leftarrow portcount[index] + 1\)
5: if \(portcount[index] \geq \tau\) then
6: \(ALERT\)
7: end if
8: end if

time maintain the advantage of fast operations [2, 3, 23, 26]. However, since access to the bitmap is a bit at a time, the only practical approach here is to use bloom filter based bitmap compression [4].

4.2 PORT-NODE-ALERT

Our approach to PORT-NODE-ALERT problem is based on the scheme for PORT-ALERT problem. In addition, we maintain counters to count the number of distinct nodes the connection attempts come from. Exact solution to find the number of distinct elements in \(m\) items require \(\Omega(m)\) space. So, we use a Bloom filter to approximate the count. The Bloom filter is used to determine whether the pair of source and destination IP addresses \((IP_1, IP_2)\) was seen earlier or not. If the pair is an unseen pair, then the corresponding count is incremented. Bloom filter allows us to use a single data structure for the entire subnet instead of using separate data structures for each machine on the subnet. This simplifies the overall design. The algorithm is given in algorithm 2.

Bloom filters can have false positives, i.e., filter returns a result indicating the data item is in the filter but actually it is not. False positive rate can be controlled by the number of hash functions and the size of the filter. We next investigate how large the bloom filter should be to accommodate a given number of users with low false positive rate. We use the parameter \(s\) for the number of
Algorithm 2 PORT-NODE-ALERT Insertion Algorithm for connection from $IP_1$ to $(IP_2, p)$

1: \(\text{index} = \text{GetIndex}(IP_2)\)
2: if \(\text{bitmap}(\text{index}, p) == 0\) then
3: \(\text{bitmap}(\text{index}, p) \leftarrow 1\)
4: \(\text{portcount[\text{index}]} \leftarrow \text{portcount[\text{index}]} + 1\)
5: end if
6: if \(\text{BloomFilter}(IP_1 || IP_2) = \text{false}\) then
7: \(\text{BloomInsert}(IP_1 || IP_2)\)
8: \(\text{nodecount[\text{index}]} \leftarrow \text{nodecount[\text{index}]} + 1\)
9: end if
10: if \(\text{portcount[\text{index}]} \geq \tau \) and \(\text{nodecount[\text{index}]} \leq \kappa\) then
11: \(\text{ALERT}\)
12: end if

connection attempts, \(k\) for the number hashes and \(m\) for the size of the filter. Ideally we want a data structure whose size depends on the number of entries \(s\). Assume that we use a bloom filter whose size \(m\) is \(\alpha s\) where \(\alpha\) is an integer denoting how much space is allocated as a multiple of \(s\). The false positive rate of the bloom filter can be expressed as

\[
(1 - (1 - \frac{1}{\alpha s})^k)^k \approx (1 - e^{-\frac{hk}{s}})^k = (1 - e^{-\frac{h}{\alpha}})^k
\]

Figure 4. False Positive Rate of Bloom Filter

(a) False Positive Rate

(b) Number of Hash Functions

False positives result in counter values to be below what they should be. Since the connection attempt appears to be in the filter, it is never added and counter is not incremented.
4.3 PORT-NODE-TIME-ALERT

Our approach to PORT-NODE-TIME-ALERT problem is based on the scheme for PORT-NODE-ALERT problem. Since solving the query exactly requires much space, we use an approximate mechanism. We divide the time into intervals of size $\Delta$ and use the scheme described in PORT-NODE-ALERT to store the data for each interval. A new data structure is maintained for each interval of size $\Delta$. Given a time threshold $\theta$, the query uses the last $R(\frac{\theta}{\Delta})$ intervals to answer the query where $R()$ denotes the round function. Given a time interval $\theta$, the difference between $\theta$ and estimated time $R(\frac{\theta}{\Delta})\Delta$ is $< \frac{\theta}{2}$. As the interval size $\Delta$ increases, the difference between $\theta$ and estimated time increases while the space requirement decreases. For older intervals only the counters need to be maintained. The bitmap and the bloom filter can be deallocated when the interval is over. Block diagram of the scheme is given in figure 5. For each old interval, $n$ counters to maintain port information and node information is needed. Number of intervals that needs to be stored is based on the time threshold $\theta$ and is given by $R(\frac{\theta}{\Delta})$. To support a large range of $\theta$ values, large number of intervals needs to be stored. Let $C_p$ denote the size of a counter to store the number of ports and let $C_n$ denotes the size of a counter to store the number of distinct machines the connection attempts come from. For each interval $n(C_p+C_n)$ space is needed to store the counters. For example, for a subnet with 256 machines and 2 byte counters space requirement per interval is 1 Kbyte which is quite small. So, interval size can be kept small (and smaller sized counters can be used).

A special set of counters to estimate the values of portcount and nodecount for a time interval of $\theta$ are maintained. These counters maintain the sum of counts for the last $R(\frac{\theta}{\Delta})$ intervals. Let $\text{portcount}_i[j]$ be the count of the number of ports for machine $j$ in interval $i$ and let $c$ be the current interval. The special counter $\text{Sumportcount}$ has the following value

$$\text{Sumportcount}[j] = \sum_{k=c-R(\frac{\theta}{\Delta})+1}^{c} \text{portcount}_k[j], 0 \leq j \leq n$$

(2)

When an interval is over, the counters are updated to reflect the new set of intervals in use. This can be done by removing the count for the interval that falls outside the window. The new interval has counts of 0 and need not be added to the sum. This can be formally stated as
\[\text{Sumportcount}[j] = \text{Sumportcount}[j] - \text{portcount}_{\frac{j}{\theta} + 1}[j], 0 \leq j \leq n \quad (3)\]

A special counter \text{Sumnodecount} is maintained to estimate the value of node-count for a time interval of \(\theta\). The process to maintain the counters is similar to \text{Sumportcount}. We leave the details due to space restrictions.

For each new connection attempt, the counters \text{portcount} and \text{nodecount} for the current interval and the special counters \text{Sumportcount} and \text{Sumportcount} are updated if necessary.

\[\text{Fig. 6. Time Difference using Fixed-sized Intervals}\]

Using fixed-sized intervals have some shortcomings. First, it is possible to bound the difference between \(\theta\) and approximated value, however it is not possible to bound the number of connection attempts missed. The time difference for several values of \(\theta\) is given in figure 6. Second, the false positive rate of bloom filter depends on the number of connection attempts in that interval. Since we don't have a bound on the number of connection attempts it is not possible to provide performance guarantees. Using variable-sized intervals solves these problems. A new interval is created when a threshold number of connections are received. Using this approach space complexity depends on the number of connections received. Using variable-sized intervals \text{sumportcount} can be updated as follows

\[\text{Sumportcount}[j] = \sum_{I_k \cap [t-\theta, t] \neq \emptyset} \text{portcount}_k[j], 0 \leq j \leq n \quad (4)\]

Maintenance of the variable \text{sumportcount} is similar to the fixed-sized interval case. Counters are updated whenever new item is added and when an interval \(I_k\) falls out of the interval \([t - \theta, t]\), the counts for the interval \(I_k\) are subtracted from \text{sumportcount}.

Counters maintained can also be used for visualization. For example, a surface plot showing \text{portcount} for a subset of the data is given in figure 7. A portscan that lasts a couple of intervals can be seen in the figure.
5 Conclusion

Although detecting port scans using a database system is possible, it requires too much space and computational overhead. In this paper, we propose space-efficient structures to detect parameterized versions of port scans. Parameters including number of ports scanned, number of source IP addresses involved in the scan and the time interval for the scan can be set by the system administrator. Proposed schemes are lightweight, require low space overhead, low computational overhead and can handle high load.

References