Implementing Real Time Port Scan Detection for the IP Backbone

Avinash Sridharan  
University of Southern California  
Dept. of Electrical Engineering  
asridhar@usc.edu

Tao Ye  
Sprint Advanced Technology Labs  
Burlingame, CA  
tao.ye@sprint.com

Sprint ATL Research Report RR07-ATL-020399

Abstract

Port scanning is prevalent in today’s Internet and often has malicious intent. Although many algorithms have been proposed for different aspects of the scan detection problem, we have seen few system discussions in the literature. Furthermore, the few existing systems are designed for enterprise gateway level Intrusion Detection. Targeting the IP backbone, we put all the pieces together in an implementation of an online port scan detecting [19] and tracking system for high speed networks. We introduce our flexible architecture, discuss trade-offs and design choices. Specifically, we go in depth to two design choices: the probabilistic counter selection and the buffer size tuning. Our choice of a simple counter is validated through an empirical analysis of trace simulation. The buffer size is derived using Lyapunov drift techniques [13] and applying the Lyapunov theorem to solve an equivalent queuing model. Although the focus of the paper is not on the measurement results of scanner tracking, we briefly summarize our key findings. Interestingly, the scanners’ behavior follows a 90-10 curve, for both active length of time and scanning rate. That is, 90% of scanners are active for a short time, with low scanning rates, while 10% are long term and fast scanners, with a few super-scanners lasting the entire duration of monitoring.

1 Introduction

Malware of many different varieties continue to spread through today’s Internet at an alarming rate and volume. Port scanning, along with email and web page phishing, are the major channels of such propagation. This perpetual and unwanted scanning traffic is often aimed at discovering and infecting vulnerable hosts with viruses, leading eventually to botnets and criminal activities [23]. To understand, detect and eliminate such traffic is a vital part of enhancing Internet security. In this paper we focus on the issue of system design for a port scan detection system for the Internet backbone.

Port scanning detection has attracted considerable attention in recent years. Many algorithms have been proposed that targets a specific aspect of very fast port scan detection: fast counting of the number of active flows [9] [24]. However, we have seen very few system discussions in the literature. These algorithms are not enough to address the practical problem of port scan detection. When they are applied to systems such as SNORT [5], where absolute thresholding is used to tag a scanner, they do not help to reduce the high false positive rates. A welcome recent development was made by Jung et al in the design and implementation of TRW [11]. TRW is a sequential hypothesis testing based scan detection algorithm, implemented in Bro [3]. It is shown to have a high detection accuracy. However, TRW is designed for the enterprise gateway level network. It relies on knowing if a source has made a failed or a successful connection, therefore requires stateful protocol analysis of the sources. This is typically only realistic for enterprise network.

We’re interested in a practical solution to scan detection in the fast transit backbone network, with a comparable high accuracy rate as TRW. Many reasons motivates us to look from the backbone point of
The main existing effort in tracking anomalies are honeypots, [22] i.e. dark addresses, and enterprise or stub network intrusion detection system (IDS) logs. The Internet Storm Center (SANS) [4] collects reports of threats from participating honeypot and IDS entities to track major threats. However, enterprise or stub networks can only observe a small portion of the scanning activity due to its limited IP address space. Honeypots also suffer from such limitation, furthermore, intelligent viruses and worms are known to evade large and well-known honeypots [6]. Our aim is to deploy detection devices in the upstream transit network that can cover a larger address space not currently monitored. It is much more difficult to evade transit networks, especially for the traffic from foreign countries. It has been shown that traffic from countries with less strict laws contributes to a large portion of scanning activities. A transit network can also observe a higher scanning rate, hence identifying a scanner faster. Finally, if scans can be accurately blocked at the top level, we can save on costly deployment and management of local IDS’s.

We argue that scan detection and containment should also be done at the transit backbone networks. To this order we build and deploy a live system to detect and track scanners in a tier 1 ISP backbone network. We present our design issues and tradeoff decisions in this work. We choose to implement TAPS [19], a connectionless port scan detection algorithm designed for fast performance and accuracy. Our software implementation is done on Linux PC’s, integrating CMON [15], an online monitoring system, and TAPS [19]. It not only performs fast scan detection at high speeds, but also tracks all activities from detected scanners. In particular, the tracking is designed to answer these questions: What are the long term behavior of scanners? Can we identify different classes of scanners?

Our contributions are four folds. First we present a flexible architecture for scanner detection on the backbone. We use a lower level passive packet monitoring system CMON [15] to classify packets into flows, and allow multiple CMON nodes to connect to our detection unit. A flow-daemon then accepts the flows and records them in a circular buffer. The TAPS detection unit processes the flows, tags scanners and tracks scanner behavior. All three logical components are stand alone software implementations and can be deployed on different hardware boxes. Second, we realize the TAPS algorithm with an online implementation. We incorporate a simple data streaming algorithm, probabilistic counting, in our implementation to make TAPS fast and memory efficient. Third, we deploy and evaluate the system performance on live backbone links. We validate the TAPS algorithm in a live performance evaluation both for its capability and accuracy. Fourth, we present a few intriguing results from tracking scanners over a duration of 3 days. We observe an interesting disparity among scanners. Namely, 90% of scanners scan at relatively low rate but 10% scan at high rates, with a few at very high rates. 90% of scanners are also active for a short time, a few hours, while 10% lasting days. A future more indepth study is planned to understand this behavior.

The rest of the paper is organized as follows. We first summarizes the key points of TAPS, the scan detection algorithm, in Section 2. Then the architecture design of our system is described in in Section 3. We follow up with the implementation, deployment details and system evaluation in Section 4. In Section 5 we discuss our observed scanner results. Related work is dpr esented in Section 6. Finally we conclude with discussion in Section 7.

2 TAPS: Time based Access Pattern Sequential hypothesis testing

At the core of our system is the online implementation of TAPS, a time-based access pattern Sequential hypothesis testing algorithm. The intuition behind this algorithm is that a scanning host’s access pattern demonstrates a high value for the ratio of \( \frac{\text{No.of destination IP}}{\text{No.of ports}} \) (simplified to \( \frac{IP}{port} \) for the rest of the text), in a given period of time. We denote this period of time as a time bin. The distribution of this ratio shows a clear demarcation in the statistical behavior of the scanner set and benign host host. This ratio is then used to perform a test for the hypothesis of whether a host is \textit{BENIGN} or a \textit{SCANNER}, across multiple time bins. A detailed description of TAPS is presented in [19] along with analytical bounds on its performance.
and an empirical evaluation of the algorithm on backbone traces. TAPS depends solely on counting the destination IPs and ports of a source, without relying on connection state information. Therefore, it can be used for detecting both TCP and UDP scans. Moreover, since backbone links are all uni-directional in nature, connection state inference is difficult. TAPS’s connectionless design make it a suitable algorithm for backbone scan detection. Finally, it is shown to be fast, reaching a decision within 6 time bins on average, while yielding low false positives and high detection rates.

Although the theoretical complexity of TAPS is low, the associated data structure plays a significant role in the performance of the algorithm. At the end of every time bin, for every source IP under test TAPS requires a count of the number of distinct IP and distinct ports accessed. A naive approach would be to maintain a heap of all the addresses seen so far. However, the time to update a heap is $O(n \log n)$ and the memory required is $O(n)$, where $n$ can be quite large. Instead, maintaining probabilistic sketches rather than exact counts can solve this problem. Research on sketches [9] [24] [10] proposes compact data structures and bitmap algorithms for flow counting with small errors bounds. Our job is to minimize the errors and its impact on false positives and negatives of scan detection. We discuss our counter of choice and the validation experiment in the following section.

2.1 Flajolet Martin Distinct Counters

A widely accepted solution to the counting problem is the Flajolet Martin distinct counters [10] (FM counters). FM counters maintain a probabilistic sketch of the incoming data and give an estimate of the number of unique elements present in the data set. The estimate could be made arbitrarily close to the actual count by increasing the number of hash functions used in the sketch. Estan.et.al [9] subsequently proposed more sophisticated multiresolution bitmap and trigger bitmap algorithms [9] that aim at a very small memory implementation (SRAM). Since our system implementation is entirely in software on common PC hardware, we choose to use the older but simpler FM counters for this design. Moreover, our empirical analysis shows that although using a small number of hash functions can decrease the accuracy of the raw count of IP or port, its effect on the accuracy of the ratio of $\frac{IP}{Port}$ is actually quite low. We proceed to describe the FM counters and our evaluation for determining the possible number of hash functions required for our implementation.

FM counters consist of $m$ hash functions and $m$ 32 bit vectors (also called the bitmap). The values generated by each hash function are used to set the bits in the corresponding vector. A bit $k$ in the $i^{th}$ vector is set for an element $x$ if $\text{min\_bit}(\text{hash}_i(x)) = k$. Here $\text{min\_bit}$ is a function that gives the least significant bit of $\text{hash}_i(x)$ that is 1. The estimate of the number of distinct elements in a multi-set $M$ is equal to $2^R\phi$, where $\phi = 0.77351$. $R$ is the position of the leftmost zero in the bitmap generated after hashing all elements in the multi-set. The only property the hash functions need to exhibit is that they can map any element $x$ from the multi-set $M$ to a value between $0 \cdot 2^L$, uniformly. Since the multi-set $M$ in question is the 32 bit Internet address space, it is relatively straightforward to design hash functions with the above property.

For example, let us assume $m = 1$, representing a single hash function. To calculate the number of distinct elements in the multi-set $M$, we need to apply the hash function to each element $x$, setting the bit pattern of one 32 bit vector based on $\text{min\_bit}(\text{hash}(x))$. The resulting vector gives us the estimation of distinct element counts in $M$. In reality, $m > 1$ and the estimate $R$ is taken as a mean of all the $m$ vectors.

2.2 Evaluating the Flajolet Martin Distinct Counters

FM counters are remarkably accurate when sufficiently large number of hash functions are used. For example, with $m = 256$ hash functions we could get estimates of the count with close to 5% standard error [10]. However, TAPS is not interested in accurate counts of distinct IP and ports; rather we are interested in a reliable estimate of whether $\frac{IP}{Port} < k$ or $\geq k$ to a threshold $k$. Even the exact value of the $\frac{IP}{Port}$ ratio is not
Table 1: Performance of FM counters in terms of % mean error for absolute values of distinct IP and distinct Port counters.

<table>
<thead>
<tr>
<th>Bin Size</th>
<th>m</th>
<th>mean error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1sec</td>
<td>256</td>
<td>0.03%</td>
</tr>
<tr>
<td>1sec</td>
<td>32</td>
<td>5%</td>
</tr>
<tr>
<td>1sec</td>
<td>8</td>
<td>15%</td>
</tr>
<tr>
<td>1min</td>
<td>256</td>
<td>0.009%</td>
</tr>
<tr>
<td>1min</td>
<td>32</td>
<td>5%</td>
</tr>
<tr>
<td>1min</td>
<td>8</td>
<td>15%</td>
</tr>
</tbody>
</table>

Figure 1: Performance of FM counters with different number of hash functions in terms of the probability of error incorrectly estimating the comparison of $\frac{IP}{Port}$ ratio with different values of $k$.

necessary. Intuitively since we are interested only in measuring the inequality, even smaller values of $m$ might still give us accurate estimates. Smaller $m$ in turn improve on the time and space complexity of our implementation in the online detection. However, accurate absolute counts are important in the tracking module for tagged scanners, therefore we use a large $m$ value.

To strengthen our intuition we performed the following empirical evaluation. We implement both data structures in TAPS, a heap and an FM counter, to evaluate FM’s accuracy. We run TAPS offline on traces that were collected on the Sprint IP backbone. At every time bin TAPS maintained the number of distinct IP and distinct ports for every source IP seen, in both data structures. We use a 1 hour trace to show results. We then compare the actual count in the heap and an estimated count from the FM counters. The above evaluation is done on the same trace for different values of $m = 8$, $m = 32$ and $m = 256$.

Table 1 presents the mean percentage error observed between the actual count and the estimated count. The mean error increases with the decrease in the number of hash functions used. It is relatively high for the $m = 8$ case ($\sim 15\%$). Thus if we required an exact count of the IP and ports accessed we would need to use $m = 256$ or $m = 32$ hash functions.

Recall that we are really only interested in a secondary value, i.e., if the $\frac{IP}{Port}$ ratio crosses a threshold. Figure 1 presents an interesting observation. It shows the probability of error in estimating the correct decision for the inequality $\frac{IP}{Port} < k$ or $\frac{IP}{Port} \geq k$. For $m = 256$ the probability of error is quite small across
all values of $k$. The probability of error increases for small values of $k$ as the number of hash functions $m$ are reduced. However even for $m = 8$ the probability of error is 0.03 for small values of $k$ ($k < 10$). For large values of $k$ the probability of error tends to zero independent of $m$. The figure 1 strengthens our intuition that even by using FM counters with small $m$ ($m = 8$) the accuracy will not significantly degrade when compared to the performance of TAPS with $m = 256$. We will quantify this argument in section 4.2.

3 Traffic Monitoring Architecture

The traffic monitoring system on which TAPS will be implemented is called CMON [15]. CMON is a real time monitoring system developed at Sprint Labs. The core components of CMON consists of a Endace DAG packet capture card [1] and a software component that can classify packets into flows and provide traffic summary in real time. The current version of the CMON has the ability to monitor backbone links up to link speeds of OC-192.

The primary challenge of performing traffic monitoring on the IP back bone is the large volume of traffic that needs to be processed every second. Secondly monitoring is not the primary task of the network and hence its imperative that any monitoring task undertaken does not affect the performance of the network. CMON addresses the second problem, by avoiding being in the critical data path and instead ‘sniffing’ packets as they go by. The first problem is then addressed by reducing the traffic volume for processing by grouping packets into flows. Analyzing flows instead of packets can reduce processing by around 20 times. Sampling or other filtering mechanisms can be activated on CMON to further reduce traffic load.

Figure 2 shows a typical CMON deployment. The CMON system is deployed by splitting a fibre cable and inserting DAG cards in the middle of the fibre link. This approach allows CMON to sniff all packets on a specific peering link. The CMON software itself consists of user configurable packet and flow level filters. In the context of this work from this point onwards we will treat CMON as a black box that has the ability to export a five tuple flow - source IP, destination IP, source port, destination port, protocol - for analysis by real time applications. A specific flow from CMON - apart from the five tuple - will also contain information about the time stamps of the first and last packet, and the number of packets that constitute the flow.

The objective of building CMON was to develop a flow capture system that can interact with monitoring
applications that could analyze flows in real time. To this end the current CMON system has the ability to export flows to a single socket. If CMON were allowed to interact with monitoring applications directly that needed to perform flow level analysis, it would adversely affect the flow capture capabilities of CMON. This in turn would require an increased sampling rate to meet the buffer constraints of CMON impacting the performance of applications such as TAPS [19] as shown by the study [14]. We there introduce a new component called **Flow Daemon** to address the main bottle neck of the system: the flow processing queue. Flow-daemon allows us to process flows asynchronously by creating a tunable buffer. This enables us to deal with large traffic variation, and ‘spikes’ or micro congestion that happen frequently in the backbone. The sole purpose of this process would be to collect flows exported by CMON, and duplicate the flows to any application that might be interested over a TCP/IP socket. The description of the integration of TAPS with CMON is done in two parts. The first part of the integration involves a description of the **Flow Daemon**. The second part would involve a description of a systems implementation of TAPS, our intrusion detection algorithm. Note although **Flow Daemon** is a generic component of CMON that will be used for integration with any application that does real time analysis of flows we specifically describe the design and implementation choices made for **Flow Daemon** to highlight the fact that the performance of the **Flow Daemon** directly impacts the performance of **TAPS**.

### 3.1 Flow Daemon

At the very least Flow Daemon is a buffer for storing flows exported from CMON. Its sole purpose is to collect flows being exported by CMON and replicating these flows to interested applications. Figure 3 shows the architecture of flow daemon.

Flow Daemon has two main components. The **Flow Collector** that listens for connections from CMON and the **Application Handler** that listens to connections from interested applications. The **Flow Collector** and the **Application Handler** share a common buffer on the hard disk. The **Flow Collector** uses a component called the **Disk Writer** to write flows to the disk and the **Application Handler** uses the **Disk Reader** to read flows from the disk.

We made a design choice of storing the flows exported by CMON on to a disk and not in volatile memory. The number of flows being exported from CMON can be potentially large and in order to cope up with the
traffic if we wanted to ensure that drop probability of flows is small, we would require a substantial amount of volatile memory. Since volatile memory is limited and expensive this would not be possible and instead of solving the problem we would have simply moved it from CMON to the Flow Daemon. Our alternative to this would be to write the flows directly out to a hard disk or a RAID. This operation is possible at the traffic speeds we desire to monitor since we are dealing with flow level semantics and not packet level semantics. Which in turn implies that the inter arrival times experienced by the Flow Daemon are considerably less compared to CMON - since it deals with packets and has to convert it to flows - allowing it to create a buffer on a hard disk or RAID device. Given that we are writing the flows out to the disk we still want to maintain space efficiency since the validity of the flows is only for the duration of a request from the monitoring application. Thus allocating a large space to buffer these flows would be uneconomical. Taking the above arguments into consideration we came up with a simple yet efficient circular buffer to be shared by the Disk Writer and Disk Reader.

The Disk Writer divides flows that it receives from the Flow Collector into time intervals of 1 minute. In order to bin flows into the specified intervals it maintains a virtual clock using the time stamps that are part of the flows. For every 1 minute it writes the flows onto a single file corresponding to that minute of data. On the disk in order to limit the size of the circular buffer we allow only a specified number of files to be created by the Disk Writer after which it would overwrite the first file it had created. The synchronization between the Disk Writer and the Disk Reader can now be achieved by implementing a centralized locking mechanism using the POSIX pthread libraries in order to enforce atomic read and write operations between the Disk Writer and Disk Reader on the files of the circular buffer. The efficiency of the buffer relies on answering the following question - What should be the size of the circular buffer? Or in other words how many minutes of data do we buffer? The validity of the question arises from the fact that despite the service rate of the application handler being greater then the arrival rate at the flow collector - this is essential to maintain rate stability - there will always be a finite bound on the buffer size required to ensure that no flows are dropped. In other words if the circular buffer is too small it would end up over writing flows before they are serviced and if they are too large they would end up wasting space.

In the following section we answer this question by modeling the system as single queue and applying Lyapunov Drift [13] techniques to calculate simple bounds on the buffer size.

3.1.1 Calculating Bounds on the Flow Daemon Buffer

Consider a single queue with the following dynamics\(^1\). Assume that the queue is slotted and works on discrete time boundaries. Every slot of time the queue receives \(A(t)\) arrivals and has \(U(t)\) backlogs in the queue. The service rate that is available to the queue is \(\bar{\mu}(t)\). Where \(\bar{\mu}(t)\) is defined as follows:

\[
\bar{\mu}(t) = \begin{cases} 
\mu(t), & U(t) \geq Q \\
0, & U(t) < Q 
\end{cases}
\]

That is the queue starts processing packets at a service rate \(\mu(t)\) only if it has a backlog \(\geq Q\). If we assume that \(\mu(t) \leq \mu_{\text{max}}\) where \(\mu_{\text{max}} \leq Q\), then the queuing dynamics can be given by the following equation:

\[
U(t + 1) = U(t) - \bar{\mu}(t) + A(t)
\]

The equivalence of the above queuing dynamics to the Flow Daemon process might now be evident. The packets in the above queuing system represent the flows in the Flow Daemon process. In the Flow

\(^1\)This is a classical queueing theoretic problem to model the effects of compression on queuing dynamics. This specific queueing problem was designed by Prof. Michael Neely at the University Of Southern California, during his lectures on Stochastic Optimization. The problem and its proof are a result of these lectures. The novelty here is the application of the results in the context of calculating the buffer sizes for Flow Daemon
In order to calculate the required buffer size we need to know the expected arrival rate and the expected service rate, \( \lambda \) and \( \mu \) respectively. By providing bounds for the back log of the queueing system we will be able to bound the buffer size of the flow daemon. A bound on the expected back log of the queue can be found by calculating the Lyapunov drift and applying the Lyapunov drift theorem. The bound on the expected back log queue will be given by:

\[
U \leq \frac{\mathbb{E}\{\mu^2\} + \mathbb{E}\{A^2\} + 2QA}{2(\mu - \lambda)} \tag{1}
\]

Where \( \lambda = EA(t) \) is the expected arrival rate for the arrival process, \( \mu \) is the expected service rate of the service process, \( \mathbb{E}\{\mu^2\} \) is the mean square of the service process and \( \mathbb{E}\{A^2\} \) is the mean square of the arrival process. The proof of the bound in equation 1 can be found in Appendix A.

### 3.1.2 Empirical Calculation of the Buffer Size

In order to calculate the required buffer size we need to know the expected arrival rate \( \lambda \), the service rate \( \mu \), the mean square of the arrival rate \( \mathbb{E}\{A^2\} \) and the mean square of the service rate \( \mathbb{E}\{\mu^2\} \). To get an estimate of the required buffer size we performed an experiment on an OC-48 link that we plan to operate on. From this point on we will refer to this link as link "SJ-21-BB-west-1". In our experiment we over provisioned our flow daemon to store 24 hours worth of data in its cyclic buffer before it starts over writing the data stored in the very first minute of the experiment. We allowed the setup CMON+Flow Daemon+TAPS to run on this link for slightly less than 24 hours. In this run we logged the number of flows that were being processed by the flow collector every second. The service rates from the logs give us a means of calculating \( \lambda \) and \( \mathbb{E}\{A^2\} \) for the arrival process at the flow collector. But in order to calculate the bound we also need to know the expected service rate, \( \mu \) and its mean square \( \mathbb{E}\{\mu^2\} \). This is quite difficult to predict since this is very specific to the system on which our monitoring application would be operational. To accomplish our task we can use the fact that the bound on the back log queue will increase as the difference between the arrival rate \( \lambda \) and the service rate \( \mu \) reduces. Thus we can approximate a loose upper bound - its equivalent to over provisioning the system - by assuming \( \mu = 1.1 \times \lambda \) and \( \mathbb{E}\{\mu^2\} = (1.1)^2 \times \mathbb{E}\{A^2\} \). Also an approximation to \( Q \) can be made by setting \( Q = \lambda \times 60 \). Using the above approximations the bound on the buffer size can be expressed in terms of \( \lambda \) and \( \mathbb{E}\{A^2\} \) alone:

\[
U \leq \frac{2.2\mathbb{E}\{A^2\} + 132\lambda^2}{0.2\lambda} \tag{2}
\]

For our specific experiment by taking averages over 2 hours of data we get \( \lambda = 11697 \) flows per second - \( \sim 12000 \) flows per second - and \( \mathbb{E}\{A^2\} = 144240416 \), giving \( \mu = 12866.7 \), \( \mathbb{E}\{\mu^2\} = 174530902 \) and \( Q = 701820 \) flows per minute. Plugging the above values in equation 2 and dividing it by \( Q \) the number of flows stored per minute we get the number of time bins we need to store data is equal to 10.91 \( \sim 11 \) time bins.

We should note that in our calculations we have made a few approximations. We assumed that the arrival process is ergodic and stationary and so is the service process. Further we have also made assumptions about the independence of the arrival and departure process. These assumptions were necessary to make the analysis tractable. But the effect of these approximation can be negated by over provisioning the system and thus taking care of any unaccounted variance in our estimation. Thus instead of storing exactly 11 minutes of data can store 1 hour data. Our experiments show that this over provisioning does not lay too much of a burden on the system. In our experiments the average disk space utilization for storing 1 hour worth of data is close to 300MB, which would be available on a standard hard disk or a RAID array. Our experience with
these setting during our evaluation of TAPS on the IP back bone suggest that these setting are conservative enough to avoid dropping any flows at the flow daemon.

3.2 TAPS

We present the design and implementation of TAPS as an application that is integrated with the CMON system. TAPS connects to the application handler of the Flow Daemon and receives flows that are exported by the CMON system. It then performs sequential hypothesis tests on a per source basis on flows received by the Flow Daemon to determine whether a source is malicious or benign. Figure 4 shows the TAPS architecture.

TAPS performs the sequential hypothesis test on every source at the end of every time bin (It's essential to note that these time bin are different than the ones used in the circular buffer for Flow Daemon. The size of the time bins and the \( \frac{IP}{port} \) ratio threshold \( k \) are user configurable parameters that are inputs to TAPS. The TAPS implementation consists of three components. The first component called the Time Manager keeps track of the information required by TAPS to perform the analysis - start and end times of flows, the sources seen in a given time bin, and the per source data namely the number of distinct IP and port that a source attempted to connect to. The second component is called the SHT Handler - Sequential Hypothesis Test Handler. This component implements the TAPS [19] algorithm. It uses the information collected by the Time Manager and performs the sequential hypothesis test at the end of each time bin. The third component Scan Monitor provides the logging functionality for every source that was tagged by the SHT Handler as a SCANNER.

Each of the three components use the Flajolet Martin distinct counters [10] to store the number of distinct IP’s and ports in a given time bin for each source. The choice of the number of hash function for the FM
Table 2: Parameters and variable that play a role in the performance of TAPS.

counters directly affects the performance of the three components as has been investigated in section 2.2

3.2.1 The Time Manager

In order to perform the test TAPS requires the following information:

- A virtual clock that demarcates the start and end of every time bin.
- A list of sources that were seen in a given time bin
- A per source data structure that keeps track of the number of distinct destination IP’s and the number of distinct ports that a source attempted to access in the given time bin.

The Time Manager is responsible for providing this information on a per time basis. The time manager maintains a virtual clock based on the time stamps of the flows. Using the virtual clock the Time Manager divides the time line into time bins that are user configurable and keeps track of the start and end of the time bins. Every time bin, it maintains a hash table with the src IP observed during the time bin acting as the key to each data entry in the table. The actual data entry is a FM counter maintaining a count of the unique IP and port visited by a source. At the end of each time bin, the time manager wakes the SHT Handler and passes the above information for it to run the TAPS [19] algorithm. A new hash table is used to store the information for the next time bin SHT Handler processes the data for the current time bin.

3.2.2 The SHT Handler

At the end of every time bin, the information about the sources observed and the number of distinct IP and ports accessed by these sources in the current bin are obtained from the Time Manager. A source is said to have generated a successful event if its \( \frac{IP}{Port} \geq k \) and it generates a failed event if \( \frac{IP}{Port} < k \), where \( k \) is a threshold. The parameters that govern the TAPS algorithm are presented in table 2. Depending on the event generated by the source the likelihood ratio is updated using the \textit{a priori} probabilities \( \theta_1 \) and \( \theta_0 \). The details of the hypothesis test can be seen in [19]. All the sources under test are maintained in a hash table and the test is run across multiple time bins. When the likelihood ratio of a source crosses the threshold \( \eta_1 \) or \( \eta_0 \) it is tagged as a SCANNER or a BENIGN host. Any source that is tagged as a scanner is then forwarded to the Scan Monitor for activating the collection of statistics on the behavior of the scanners.

3.2.3 The Scan Monitor

Whenever the SHT Handler tags a source as a scanner it forwards the source IP to the scan monitor. The scan monitor maintains the same time bins as the Time Manager with the help of a virtual clock derived from the time stamps associated with flows being exported by the Flow Daemon. For every time bin it maintains the following statistics for each source IP that has been tagged as a scanner:
### Table 3: Description of backbone links that our initial deployment monitors

<table>
<thead>
<tr>
<th>Link name</th>
<th>Description</th>
<th>Utilization (kpps)</th>
<th>Utilization (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB-west-1</td>
<td>peering link incoming</td>
<td>72</td>
<td>320</td>
</tr>
<tr>
<td>BB-west-2</td>
<td>peering link outgoing</td>
<td>115</td>
<td>560</td>
</tr>
</tbody>
</table>

- The number of distinct IP and ports accessed by the scanner
- The number time bins that the scanner has been active for
- The first time bin at which the source in question was tagged as a scanner.

The statistics are maintained using the FM counters. The logs for the suspected scanners can be archived over days on a RAID array. Apart from assisting in the evaluation of TAPS the objective of logging the above data is to provide tools higher up in the hierarchy to do asynchronous analysis in generating alarms and statistics for a larger network management system.

## 4 System Evaluation

In this section we detail the system deployment and performance evaluation.

### 4.1 Deployment

We deploy the entire system to monitor two backbone OC-48 links, described in 3. These links represent a well utilized peering point of Sprint’s backbone network with another tier 1 ISP, in the west coast of United States.

A CMON [15] passive monitoring node is deployed on a Linux PC equipped with DAG 4.3 optical packet capture card [1]. We allocate one PC per link, due to the capability of the capture card. We instrument CMON to start one stream to do flow classification on the packet input. Flows are classified by the 5-tuple (source IP, destination IP, source port, destination port and protocol) definition, with an active timeout period of 1 minute. Flows generated from both CMON nodes are exported through sockets to a third PC. On this PC Flow-daemon and TAPS run as separate processes. These PCs have a typical 2U rack server configuration, with dual processors of 2.5Ghz speed, 4GB of memory and 350GByte of hard disk space. They run Linux with a 2.6 kernel from the Fedora Core 5 distribution.

Our buffer unit for flow-daemon is 1 minute. Flow-daemon witters the flows to 60 units of circular buffer in plain file on disk, with each minute of flow written as one file. The minute buffer unit files are over-written after 1 hour. TAPS then processes these buffer files and tag scanners.

A common mode of failure is packet or flow dropping, indicating the system can not process as fast as traffic arrives. With this setup we observed no packets or flow dropping at any of the components, for varies running times. The longest test run time is 12 days. This indicates that the systems function well under this traffic load. We can detect this common mode of failure at three locations. Packet drops can be detected at the DAG card and an alert will be send by CMON. Flow drops can be detected by flow-daemon if the writing speed can not keep up with the arrival of flows at the socket. Finally TAPS’s time manager can also signal an alarm if it cannot process the flows in the buffer in time.

### 4.2 Number of hash functions and accuracy

In order to evaluate the affect on accuracy we ran CMON+Flow Daemon on an OC-48 link. Three different versions of TAPS, with different number of hash functions ($m = 8, m = 32, m = 256$) were made to
<table>
<thead>
<tr>
<th>$m$</th>
<th>missed</th>
<th>mean</th>
<th>variance</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>18 \ 93</td>
<td>2.3226</td>
<td>2.224</td>
<td>1.49</td>
</tr>
<tr>
<td>8</td>
<td>15 \ 93</td>
<td>3.3187</td>
<td>6.0638</td>
<td>2.46</td>
</tr>
</tbody>
</table>

Table 4: The mean, variance and standard deviation of the $\frac{IP}{port}$ ratio for the scanners missed by TAPS\_8 and TAPS\_32 and caught by TAPS\_256.

connect with the Flow Daemon which was receiving flows from CMON. We denote these as - TAPS\_256, TAPS\_32 and TAPS\_8. All three versions of TAPS were started simultaneously and the system was allowed to run for an approximate duration of 12 hours. The parameter setting for all three versions of the TAPS was $k = 3, \theta_1 = 0.2, \theta_0 = 0.8, \eta_1 = 99$ and $\eta_0 = 0.01$. In the 12 hour duration TAPS\_256 caught the maximum number of scanners (96), followed by TAPS\_8 (87), followed by TAPS\_32 (80). As is evident the number of scanners tagged by the three different versions of TAPS is very similar. From our empirical evaluation of the Flajolet Martin distinct counters in section 2.2 we can assume that a hash function size of $m = 256$ is accurate and hence we assume the set of scanners generated by TAPS\_256 to be the ground truth for comparison with TAPS\_32 and TAPS\_8. For our comparison we are interested in understanding the scanners that were caught by TAPS\_256 and missed by TAPS\_8. For TAPS\_32 there were 20 scanners that were caught by TAPS\_256 but were missed by TAPS\_32. For TAPS\_8 there were 17 scanners that were caught by TAPS\_256 that were missed by TAPS\_8. Of the missed scanners on closer observation, from our logs we could see that two of the scanners lasted for a duration of less then 6 seconds. Note the logs for each scanner are generated ones they have been tagged by the SHT Handler. We have therefore neglected these two scanners while analyzing the behavior of scanners missed by TAPS\_32 and TAPS\_8.

Table 4 shows that the mean of the $\frac{IP}{port}$ of the scanners missed by TAPS\_8 and TAPS\_32 are close to the threshold $k = 3$, implying that these are relatively low rate scanners. Further the mean, variance and the standard deviation of the scanners missed by TAPS\_32 are smaller than those missed by TAPS\_8. This is an expected behavior and can be inferred from figure 1. The table implies that the scanners missed by using a smaller hash function value ($m = 8 / m = 32$) are low rate scanners. Therefore the choice of using a smaller value for $m$ should depend on the necessity of detecting low rate scanners. From the perspective of the back bone our belief is that detecting high rate scanners would be more crucial is stopping worm spreads and the task of accurate detection of low rate scanners can be pushed towards the edges. This motivates the argument for using hash functions of small values $m = 8$ at the back bone.

4.3 Scanning Rate

Scanning rate is a metric that is used by TAPS to characterize the behavior of a scanner. The scanning rate of a source under test is the expected value of $\frac{IP}{port}$ ratio exhibited by the source every time bin. Quantitatively the scanning rate is given by :

$$scanning\ rate = \frac{E[\frac{IP}{port} | H_1]}{E[N|H_1]}$$

Where $E[IP|H_1]$ is the expected number of IP’s accessed by a source given that it is a scanner and $E[N|H_1]$ is the expected number of time bins required by TAPS to come to a decision that the source in question was a scanner. In [19] the authors had given an analytical lower bound on the scanning rate that can be detected by TAPS:

$$scanning\ rate = \frac{1}{E[N|H_1]} \times \left(\frac{k + \theta_1 \theta_0}{\theta_0} \right) \left(\frac{\ln(\eta_1)}{\ln(\frac{1 - \theta_1}{1 - \theta_0})} - 1\right)$$

Where the parameters $k, \theta_1, \theta_0, \eta_1$ and $\eta_0$ have been defined in section 2.
The lower bound thus provides TAPS with a threshold above which every scanner would be detected. The parameter $k$ which is the threshold on the IP/port ratio used by TAPS to classify a source as a SCANNER or a BENIGN host every time bin for its sequential hypothesis test provides the operator with a knob to control the scanning rate that will be detected by TAPS. Further this threshold also provides the operator with the ability to minimize the false positives. For example by setting the scanning rate very high the operator can observe very fast scanners and neglect the slow scanners. This feature is quite relevant to port scan detection at the back bone where it might more prudent to catch fast spreading worms rather than trying to catch every source that is trying to perform a scan. Further if a scanner tries to evade TAPS the only way this is possible is by applying a scanning rate that is below the TAPS scanning rate threshold. This in turn would throttle the rate at which a scanner can spread.

The objective of this section is to evaluate the performance of TAPS with respect to its detection capabilities given a specific value of the scanning rate threshold. The two factors that could adversely affect the performance of TAPS in terms of the number of false positives it generates are flow sampling and an inaccurate count of the number of distinct IP’s and ports visited by a source every time bin. By presenting the CDF of the scanning rate of the scanners we highlight the fact that our implementation of Flow Daemon and our choice of the Flajolet Martin Distinct Counters for maintaining an estimate of the count for the number of distinct IP and ports thus guaranteeing a good performance from TAPS.

For our experiments we chose two links, an OC-48 link and an OC-192 link. Both the links are peering links in the Sprint Link IP back bone. Two CMON systems were attached to each of these links and a separate TAPS process was attached to each of these two CMON processes with the help of a Flow Daemon in order to perform real time port scan detection. The parameters chosen for TAPS on both these links were as follows; $\theta_0 = 0.8, \theta_1 = 0.2, \eta_1 = 99, \eta_0 = 0.01$. We ran TAPS with two different values of $k = 3$ and $k = 5$ on the two CMON systems mentioned above. By using equation 3 the values of scanning rate that we get corresponding to $k = 3$ is, $scanning\ rate = 1.77$ and for $k = 5$, $scanning\ rate = 2.75$.

For each of the two links we observed the traffic for 24 hours. The number of scanners tagged by TAPS
for each of the two links is presented in table 5. A simple fact to be noted is that as the threshold is increased
the number of scanners detected decreases which is one the basic checks that is essential to the functionality
of TAPS. Figures 5(a) and 5(b) present the CDF’s for the scanning rate for scanners tagged by TAPS on both
the links for different $k$ values. Before describing the characteristics of the CDF we would like to present
the methodology used to calculate the expected scanning rate for each of the scanners. For each scanner that
was detected we started observing the $\frac{IP}{port}$ ratio every time bin from the point at which it was detected. The
total value of the $\frac{IP}{port}$ - summation of the $\frac{IP}{port}$ ratio - is then divided by the number of active time bins in the
first hour from the point of detection. Quantitatively:

$$scanning\ rate = \frac{\sum_{i=0}^{T-1} \frac{IP(i)}{port(i)}}{T}$$

Where $T$ is the total number of active time bins within the first hour of detection and $IP(i)$ and $port(i)$
are the number of distinct IP and port’s accessed by the scanner in the $i^{th}$ time bin. The reason for observing
only the first hour is the stationarity of the scanner process. The analysis and calculation of the bounds
assume that the scanner is a stationary process that would exhibit the same behavior for a very large value of
$T$. However in practice this is not possible since human intervention and network dynamics might always
shut down the source thus making the scanner process observed non stationary. In other words in practice
the stationarity of the process holds only for a limited time after the scanner is detected. Thus by observing
the expected behavior of the process for only the first hour we hope to capture expected steady state of the
scanner that is predicted. We concede that this is not a perfect technique and network dynamics and human
intervention can skew the stochastic of the process even within the first hour. But we have to function within
the domains of realism and thus at this point we believe this is the best estimate we can provide.

However the good fortune is that the CDF’s highlight that we are able to capture the expected behavior
of TAPS with regard to detection and scanning thresholds. The CDF’s indicate that there are only a small
percentage of scanners below the scanning threshold of $k = 3$ and $k = 5$. These figures might be skewed
due to the non-stationarity of the low rate scanner as well.

To highlight the performance of TAPS it should be noted the CDF of the $k = 5$ is lower then the CDF of
$k = 3$. This is the expected behavior since by increasing the scanning rate threshold you would filter quite
a few of the low rate scanners detected by a lower scanning rate threshold. Further the lowest scanning rate
detected in the $k = 3$ case is smaller then the lowest scanning rate detected by the $k = 5$ case. Further we
would like to point out that in both the figures 5(a) and 5(b) we haven’t presented the complete range of the
x-axis. The reason being that there were scanners detected with a scanning rate > 100. By including them
in the plot we would not be able to highlight the performance of TAPS in avoiding scanners below the set
threshold. Although it should be noted that all scanners were considered while calculating the CDF.

### 5 Scanner Profiling Results

We discuss our initial results that have been compiled from the logs of the **Scan Monitor**. Once a source
is tagged as a scanner, the tracking component records the $\frac{IP}{port}$ ratio, the number of distinct IP, the number
of distinct port every time bin that the scanner was observed. Since we require an accurate estimate of the

<table>
<thead>
<tr>
<th></th>
<th>OC-48</th>
<th>OC-192</th>
<th>scanning threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 3$</td>
<td>scanners=367</td>
<td>scanners = 293</td>
<td>1.77</td>
</tr>
<tr>
<td>$k = 5$</td>
<td>scanners = 265</td>
<td>scanners = 218</td>
<td>2.75</td>
</tr>
<tr>
<td>False Positives</td>
<td>15%</td>
<td>18%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: The number of scanners detected for different setting of $k$ on the OC-48 and the OC-192 links.
The first question we ask ourselves is: What is a typical scanning rate on the backbone? Interestingly, we observe that scanners scan at very different rates. From one hour of data, in Fig 5(a), we can see that 80% of scanners scan at a relatively low 9 IP/port per second, while 5% scan at a much higher rate of 48 IP/port per second. There are also close to 2% scanners that scan at over 500 IP/port per second. This is not shown in Fig 5(a) due to the limit range of x-axis graph.

Second, we are interested in the length of the active time of the scanners. Fig 6(a) shows the distribution of active time on the BB-west-1 link. 90% of scanners are active for less than 5.5 hours, while a few super-scanners are active for ∼ 3 days that we collected data for. BB-west-2 link shows a similar trend, with 90% of scanners active for 5.5 hours or less, and again, a few super-scanners active the entire time. Many factors affect the active time we observe. It is possible that most scanners move to other areas of the Internet after a period of scanning in Sprint’s network. Yet the same distribution exists for the entire footprint covered by a scanner. For BB-west-1 in Fig 7(b) 90% of the scanners only cover less than 100000 IP addresses in our observation, while a few super-scanners cover 3.5 million IP addresses in three days.

It is interesting to note that most scanners are transient, relatively low rate and cover a small number of IP addresses, while a few super-scanners scan for long term, are high rate and cover large footprint area. More study is needed to understand this behavior.

We now proceed to divide the scanners into two categories: persistent scanners and transient scanners, and show other differences of their behaviors. We set the division point at 5.55 hour of active time, i.e., the 90-10 divide point. Fig 8(b) shows the ports scanned by the scanner active for more than 5.55 hours and Fig 8(a) for scanners active for less than 5.55 hours. Firstly there are some overlaps, such as port 135 and 139. Although it is not possible to detect a specific type of worm by looking at the port itself, this specific observation suggests that the malware generating this particular port profile might be a known worm (possibly the Blaster worm). The most scanned port in the two sets tells a different story. The most scanned port is different in the two sets, suggesting that the most aggressive malware in the two categories might be different.

In figures 9(a) and 9(b) we present the histogram of the No. of IP’s accessed for two sets of scanners. A point that might not be highlighted in the figure but is intriguing is that in figure 9(b) there 3 scanners that have accessed more then $10^6$ IP addresses. Note that these scanners have accessed these IP addresses in less than 5.5 hours.
Figure 7: The CDF of the total number of active time bins and the PDF of the distinct number of distinct IP’s accessed by the scanner for the BB-WEST-2 link.

Figure 8: Port profile based on Active Time Bins for the BB-west-2 link scanners.
Figure 9: Classification based on Active Time Bins for the BB-west-2 link scanners, active time bins ≥ 5.5hrs.

We acknowledge that these results are preliminary. These results have been presented here to highlight the tracking feature of the Scan Monitor component in TAPS. The ability to catalog the behavior of scanners in real time at the backbone is potentially quite powerful. In a future more complete study, we hope to show the relationship between high rate, long running and large footprint scanners by running our system over longer periods of time and on different links.

6 Related Work

Jung et al [11] has proposed TRW to detect scanners and has implemented it on Bro [3]. The intuition of TRW is based on the notion of a “failed connection”, i.e. scanners make many more failed connections than a benign host. The limitation is that bi-directional traffic is needed to make the inference of a “connection”. Also the application of the notion of a “connection” to UDP scans becomes cryptic. However, Jung.et.al were one of the pioneers in using sequential hypothesis testing for port scan detection and it has thus influenced the design of TAPS [19].

The class of algorithms BeProf [21], [20] are designed for backbone traffic, although their objective is to profile general anomalous behavior that deviates from normal traffic behavior, using datamining techniques to extract significant clusters. The parameter tuning of BeProf is less than intuitive and can be hard to master.

In the domain of probabilistic sketches [9] [24] and [8] proposed more recent probabilistic counting algorithms than the Flajolet Martin Probabilistic counter (FM counter) we have used. Their focus is on reducing memory in order to design ASICS that could perform detection in hardware. However as shown in section 2.2 for our purposes the FM counter with a small amount of memory (8 hash functions) performs close to FM counters with a larger number of hash function.

In term of working implementations, earlybird [18], Autograph [12] are signature extraction systems. Gigascope [2] is a general purpose application level monitoring system with database querying capabilities. Snort [5], Bro [17] [3] and other commercial enterprise IDS/IPS are deployed to detect and block scans at the enterprise gateways.

Many studies have characterized traffic received by dark addresses, termed “background radiation”, [16] observed from many Honeypots. [16] classified and characterized many anomalies, and is an inspiration to
our work. In a landmark study, the authors of [7] model the location of the sources of background radiation with a multi-fractal model and conclude that they form dense clusters. In our work we design a system to first detect the scanners from the wide area network, then track the long term behavior of sources across time. We view traffic from a different angle, i.e. operational network rather than dark spaces. The two approaches compliment each other.

7 Discussion

We present a system implementation and deployment of an online port scan detection algorithm aimed at the IP backbone. Previous works have concentrated either on a certain aspect of the problem (counting flows), or are targeted towards enterprise networks. We use probabilistic counters from the domain of data streaming [10] and queueing theoretic analysis to bound the buffer size to make a practical implementation of our online detection algorithm TAPS [19] possible. Our implementation is designed to operate on Linux based PC’s instead of special purpose hardware. It is scalable since each component is stand alone and can interact with other components using a standard TCP/IP stack, allowing the system to be distributed or centralized depending on the requirements of the deployment.

We’re in the process of collecting long term - to the order of weeks - traces of scanners. In order to profile their behavior for future studies. There are still aspects of the system that pose considerable design challenges. For e.g. currently we are tracking all scanners tagged for the entire duration of the setup. Being an experimental setup this is acceptable however in a real deployment we need to have metrics defining time periods when scanners can be pruned from the monitoring tree. Further there are always false positives that are generated. This is a result of the uncertainty in defining the ground truth. We could use techniques from signal and image processing to do post mortem analysis on the data being collected by the Scan Monitor to make the system more robust and efficient.

References


A Bounding the Expected Queue Backlog

The queueing dynamics are given by:

\[ U(t+1) = U(t) - \mu(t) + A(t) \]
Using the above queueing dynamics we can get the inequality:

\[
U(t + 1)^2 \leq U(t)^2 + \tilde{\mu}(t)^2 + A(t)^2 - 2U(t)[\tilde{\mu}(t) - A(t)] \\
\leq U(t)^2 + \mu(t)^2 + A(t)^2 - 2U(t)[\mu(t) - A(t)] \\
= U(t)^2 + \mu(t)^2 + A(t)^2 - 2U(t)[\mu(t) - A(t)] + 2U()[\mu(t) - \tilde{\mu}(t)] \\
\leq U(t)^2 + \mu(t)^2 + A(t)^2 - 2U(t)[\mu(t) - A(t)] + 2Q\mu(t)
\]

The final inequality follows since \(\tilde{\mu}(t) \leq \mu(t)\) for all \(t\), thus \(2U(t)[\mu(t) - \tilde{\mu}(t)] \leq 2Q\mu(t)\) for all \(t\). Taking conditional expectations of the above and assuming the random variables to be i.i.d the conditional Lyapunov drift is given by:

\[
\delta(U(t)) \leq \mathbb{E}\mu^2 + \mathbb{E}A^2 - 2U(t)[\overline{\mu} - \lambda] + 2Q\overline{\mu}
\]

Under rate stable conditions (\(\lambda < \overline{\mu}\)), applying the Lyapunov drift theorem [13]:

\[
\overline{U} \leq \frac{\mathbb{E}\{\mu^2\} + \mathbb{E}\{A^2\} + 2Q\overline{\mu}}{2(\overline{\mu} - \lambda)}
\]