A practical approach to portscan detection in very high-speed links

Jakub Mikians*, Pere Barlet-Ros, Josep Sanjuàs-Cuxart, and Josep Solé-Pareta

UPC BarcelonaTech, Barcelona, Spain
{jmikians,pbarlet,jsanjuas,pareta}@ac.upc.edu

Abstract. Port scans are continuously used by both worms and human attackers to probe for vulnerabilities in Internet facing systems. In this paper, we present a new method to efficiently detect TCP port scans in very high-speed links. The main idea behind our approach is to early discard those handshake packets that are not strictly needed to reliably detect port scans. We show that with just a couple of Bloom filters to track active servers and TCP handshakes we can easily discard about 85% of all handshake packets with negligible loss in accuracy. This significantly reduces both the memory requirements and CPU cost per packet. We evaluated our algorithm using packet traces and live traffic from 1 and 10 GigE academic networks. Our results show that our method requires less than 1 MB to accurately monitor a 10 Gb/s link, which perfectly fits in the cache memory of nowadays' general-purpose processors.

1 Introduction and Related Work

Every day both individuals and companies depend more on the reliability and safety of Internet connections. However, even today, entire industry branches or countries can be a target of an attack (e.g., `Stuxnet` [3]). Most attacks start with a recognition phase, where an attacker looks for attack vectors in one or several victim systems. Port scanning is arguably the most widely used technique by both worms and human attackers to probe for vulnerabilities in Internet systems.

Given the large implications in network security, several previous works have addressed the problem of how to efficiently and reliably detect port scans. Most proposed solutions require tracking individual network connections (e.g., [6, 15, 14]). This approach however does not scale to very high-speed links, where the number of concurrent flows can be extremely large. For example, a naive solution based on a hash table would require large amounts of DRAM (e.g., to store flow identifiers) and several memory accesses per packet (e.g., to handle collisions). Nevertheless, access times of current DRAM technology cannot keep up with worst-case packet interarrival times of very high-speed links (e.g., 32 ns in OC-192 or 8 ns in OC-768 links).

Traffic sampling is considered as the standard solution to this problem. Unfortunately, recent studies [7, 5] have shown that the impact of sampling on

* J. Mikians' work was partially supported by an FI grant from Generalitat de Catalunya.

portscan detection algorithms is extremely large. Another alternative is the use of probabilistic, space-efficient data structures, such as Bloom filters [16, 11], which significantly reduce the memory requirements of detection algorithms. This way, the required data structures can fit in fast SRAM, which has access times below 10 ns. Although we are not aware of any survey paper covering the use of Bloom Filters for portscan detection, [11, 15] provide a good overview on the work in this area.

In this paper, we present a practical method to detect TCP port scans in very high-speed links that follows this second approach. A key assumption behind our method is that, apart from data traffic, we can even discard most TCP handshake packets and still be able to successfully detect port scans.

First, we ignore legitimate handshakes using a *whitelist* of active server IP-port pairs. Second, we discard those failed connections that do not correspond to scans, such as TCP retransmissions, packets from other network attacks (e.g., SYN floods) or configuration errors (e.g., P2P nodes down or misconfigured domain servers). In order to discard handshake packets, we use two Bloom filters. Surprisingly, we show that this simple solution can drop about 85% of all handshake packets with negligible loss in accuracy. This significantly reduces the number of memory accesses, CPU and memory requirements of our algorithm.

After filtering most part of the traffic, we still need to track the number of failed connections for the remaining sources. Although there is a potentially very large number of active sources, most of them will fail very few handshakes, while scanners will fail many. Thus, the detection problem can be seen as the well-known problem of finding the top-k elements from a data stream [8]. In order to efficiently detect port scans, we use an efficient top-k data structure based on the *Stream-Summary* proposed in [9], which has a constant memory usage.

We evaluated our algorithm in 1 and 10 GigE academic networks [1]. Our results show that our method requires less than 1 MB to accurately monitor a 10 Gb/s link. Therefore, it can be implemented in fast SRAM and integrated in router line cards, or reside in cache memory of general-purpose processors.

The rest of this paper is organized as follows. Sec. 2 describes our portscan detection algorithm in detail. Sec. 3 evaluates the performance of the algorithm with both packet traces and live network traffic. Finally, Sec. 4 concludes the paper and outlines our future work.

2 Detection Algorithm

Port scans are characterized by a simple feature: they attempt to connect to many targets but only get few responses. This imbalance in the number of attempts and successes is the basis of several portscan detection algorithms. A portscan detection algorithm can then be divided into two different problems: (1) detecting failed connections, and (2) tracking the sources responsible for them. Both (1) and (2) are challenging in high-speed networks, since they require a significant amount of memory and computing power to process packets

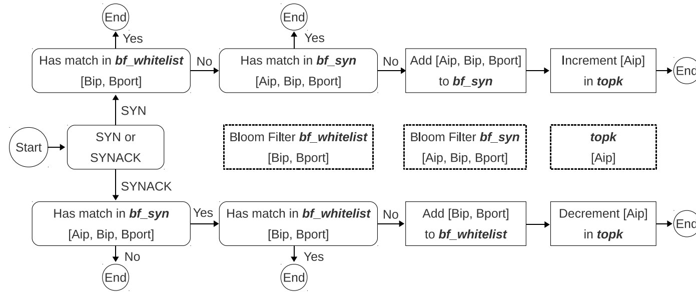


Fig. 1: Algorithm description.

at line speed. As already discussed in Sec. 1, a naive solution based on a hash table is impractical in this case, although it can be used in small networks.

In this section, we present a practical solution that copes with these two problems by reducing both the volume of processed traffic and the memory requirements of the detection algorithm. In Sec. 2.1, we describe a simple method to discard unnecessary traffic using Bloom filters, which significantly simplifies problem (1), while Sec. 2.2 concentrates on identifying scanners using a lightweight counting structure that addresses problem (2).

For the sake of clarity, throughout this section, we will refer to the client host that initiates the handshake as A, with IP address A_{ip}, and to the server that receives the connection as B, with address B_{ip} and port B_{port}.

2.1 Detecting Failed Connections

We can define a failed connection as one for which a client does not get a *SynAck* response from the server after having sent the corresponding *Syn* packet. Therefore, to detect failed connections, we can ignore data traffic and focus only on *Syn*/*SynAck* packets. According to our traces (described later in Sec. 3), these control packets represent only 1.5% of all TCP traffic.

In addition, we can ignore legitimate handshakes to detect port scans, given that a scanner will always fail a large number of connections compared to a normal host. In order to efficiently discard connections directed towards a working service, we can use a Bloom filter that maintains a whitelist of active server IP-port pairs (*bf_whitelist*). In particular, for every new *SynAck* response, we add the tuple $[B_{ip}, B_{port}]$ into this Bloom filter.

Since we are especially interested in those clients that connect to many unique destination addresses and ports, we can also discard those repeated connection attempts to the same destination. Besides standard TCP retransmissions, many applications try to reconnect several times (even hundreds) to the same destination after a failed connection (e.g., P2P nodes, misconfigured proxies, mail servers or VPN applications). Surprisingly, repeated *Syn* packets are extremely common according to our traces (see Sec. 3). In order to efficiently drop duplicated *Syn* packets to the same destination IP-port pair, we use a second Bloom

filter (*bf_syn*). For every *Syn* packet observed, we store the tuple $[A_{ip}, B_{ip}, B_{port}]$ in the Bloom filter. As we will see later, using this second filter has the additional advantage of protecting the *bf_whitelist* from being saturated by many *SynAck* packets sent by a malicious user (i.e., *SynAck* packets are ignored if they are not an answer from a previous *Syn*).

Although Bloom filters can have false positives, they have a negligible impact on our method as we show in Sec. 3. In addition, in case that one or both filters get saturated (e.g., if they are not properly dimensioned), the algorithm will produce False Negatives instead of False Positives, which is an important feature for systems automatically blocking port scanners [16].

Fig. 1 presents our algorithm in detail. After a packet arrival, we check if it is a *Syn* or a *SynAck* packet. Otherwise, the packet is dropped. In case it is a *Syn* packet, we check if the $[B_{ip}, B_{port}]$ tuple corresponds to a known destination in the *bf_whitelist*. In this case, the packet is directly dropped. If not, we check if it is a repeated connection attempt in the *bf_syn* filter. In this case, the packet is also dropped. Otherwise, the $[A_{ip}, B_{ip}, B_{port}]$ tuple is stored in the *bf_syn* filter and the A_{ip} source is incremented in the counting structure (described later in Sec. 2.2). For a *SynAck* packet, we first check if it is a response from a previous *Syn* packet in the *bf_syn* filter. Otherwise, the packet is dropped. Next, we check if the $[B_{ip}, B_{port}]$ tuple is already in the *bf_whitelist*. If not, the destination $[B_{ip}, B_{port}]$ is stored in the whitelist and the $[A_{ip}]$ source is decremented. Therefore, we use the *bf_whitelist* for two different purposes: (*i*) to keep track of active destinations, and (*ii*) to check if a source needs to be decremented after the connection has been established.[1]

2.2 Identifying Scanners

The algorithm described in Sec. 2.1 produces a series of increments and decrements for new connections and completed handshakes respectively. From this sequence, we want to identify the most active producers of failed connections, which will very likely correspond to port scanners. This can be seen as the well-known problem of identifying the top-k most frequent elements in a data stream.

For this purpose, we need a data structure that has limited memory usage and supports both incrementing and decrementing. Fortunately, the recent literature provides us with several efficient top-k algorithms [8]. From those, we selected the *Stream-Summary* data structure [9], since it uses a constant (and small) amount of memory. However, our algorithm is not bound to a particular top-k data structure. Although the original *Stream-Summary* does not support decrementing, we made a straightforward extension to support a limited number of decrements. We called this extension *Span-Dec*. As we will see in Sec. 3, in the

[1] Note that using *bf_whitelist* to check which decrements are needed can introduce errors of 1 unit in the counting structure if several *Syn* packets from different sources are sent to an active destination before it enters the whitelist. Although this unusual situation cannot be exploited by an attacker, it could be easily solved by adding a filter similar to *bf_syn* for *SynAck* packets.

particular context of portscan detection, the data structure behaves almost like an ideal hash table, but using much less memory. Although the particular implementation details of the top-k data structure are not essential to understand our algorithm, for the sake of completeness, we include below a short description of both mentioned structures.

Stream-Summary. This structure is part of the *Space-Saving* algorithm [9] that finds the most frequent elements in a data stream. It is able to observe up to $elem_{max}$ distinct elements at once. Every element e_i has an assigned counter cnt_i. All counters with the same value are linked into the same bucket. The buckets are linked together and they can be dynamically created and destroyed. When an element e_i is incremented, it is detached from its bucket and attached to a neighbor bucket with the new value. When the maximum number of observed elements ($elem_{max}$) is reached, a new incoming element evicts the element with the smallest counter. Each element has a maximum overestimation ε_i that depends on the value of the evicted element. The element frequency is estimated as $freq(e_i) = cnt_i - \varepsilon_i$. The algorithm is lightweight and it requires only $\frac{1}{\epsilon}$ counters for a specified error rate ϵ. See [9] for a more detailed description.

Span-Dec. The original *Stream-Summary* does not support decrementing. However, we need to discount those established connections for which the corresponding *Syn* has passed both Bloom filters. Therefore, we made a simple modification to the original *Stream-Summary* to support a limited number of decrements. In particular, instead of having a single counter per element, we use two counters: $cnt_L(e_i)$ and $cnt_H(e_i)$. We also specify a maximum allowed difference between both counters $span_{max}$, which controls the tradeoff between the number of allowed decrements and the error ε_i of the estimate. When an element is incremented, $cnt_H(e_i)$ is moved as in the original *Stream-Summary*. In case that the difference between both counters is greater than $span_{max}$, the $cnt_L(e_i)$ is also incremented. In order to decrement an element e_i, the $cnt_H(e_i)$ is decremented, but never below the value of $cnt_L(e_i)$. This solution can be understood as an “undo” operation, where $span_{max}$ is the “undo” depth. The frequency of an element e_i is estimated as $freq(e_i) = cnt_H(e_i) - \varepsilon_i$. The technical report [10] provides a detailed description of this extension.

As shown in Fig. 1, our detection algorithm uses *Span-Dec* to maintain the count of failed connections per source $[A_{ip}]$. This solution is useful to detect both horizontal and vertical port scans. However, if we are interested only in a particular type of scan, we can use instead $[A_{ip}, B_{port}]$ to detect horizontal port scans and $[A_{ip}, B_{ip}]$ to detect vertical ones.

3 Results

In the evaluation we used four traces. `trace A` was captured from the 1GigE access link of UPC, which connects about 50,000 users. `trace A0` is a modified version of `trace A` that we describe later. `trace B` was taken from the MAWI

Table 1: Statistics of the traces. `trace C` only accounts for *Syn*/*SynAck* packets.

	`trace A` 30min @ 1GigE	`trace B` 2h @ OC-3	`trace C` 30min @ 10GigE	`trace A0` 30min @ 1GigE
date	2010-05-18	2010-04-16	2010-07-29	2010-05-18
TCP packets	228,848,927	144,885,865	13,978,845	97,380,742
TCP sources	188,136	263,055	467,264	89,086
TCP flows	2,892,334	5,199,928	11,526,323	1,133,392
average usage	879.1 Mb/s	185 Mb/s	3.5 Gb/s	n/a

Working Group Traffic Archive [2]. `trace C` was captured from the 10GigE link that connects the Catalan Research and Education Network to the Internet. This link connects more than seventy universities and research centers. Due to the link speed, for `trace C` we only collected *Syn*/*SynAck* packets. Statistics of the traces are presented in Tab. 1. We published all the packet traces used in this work, with anonymized IP addresses, at [1].

For the evaluation, we needed a ground truth trace to check if a detected scanner was a real scanner or a (misclassified) legitimate source. For this purpose, we modified `trace A` by removing all real scanners. We scanned the trace using Bro [12] with both its standard algorithm and the TRW algorithm. Although Bro is an online tool that does not guarantee an accurate ground truth, we used a low alarm threshold (25) and removed all the flows from the reported IP addresses to make sure that no scanning traffic is left, even if some legitimate traffic was also removed. Later, following the methodology proposed in [11], we injected artificial scans to build a ground truth: 1000 scanners with success ratio 0.2 and 1000 benign sources with success ratio 0.8. The interval between *Syn*-*SynAck* packets was taken uniformly from the range $(0, 450ms)$, while the backoff time between *Syns* was modeled using an exponential distribution [11]. All modifications resulted in `trace A0` that serves as the ground truth for Sec. 3.1. Traces `B` and `C` were not modified.

3.1 Evaluation

This section covers the evaluation of our algorithm. First, we present an example of how it is dimensioned. Next, we check the performance and validate its accuracy with packet traces. Finally, we deploy it in an operational 10 GigE link.

Dimensioning. We followed a conservative approach to handle an unexpected growth of traffic or peaks. For *bf_whitelist*, we checked the mean number of distinct $[B_{ip}, B_{port}]$ tuples in the trace, multiplied this value by 3 and we assumed a maximum collision probability of $p_{coll} = 0.01$. We used an arbitrary length of the measurement window of 2 minutes. Although in this paper we do not evaluate this parameter, its value is important. As the filters are reset at the end of every period, the window size represents a tradeoff between the memory usage of the algorithm and its ability to detect slow scanners. With those values, we calculated the optimal size of the Bloom filter. We repeated the procedure for

Table 2: Configuration parameters for the evaluated traces.

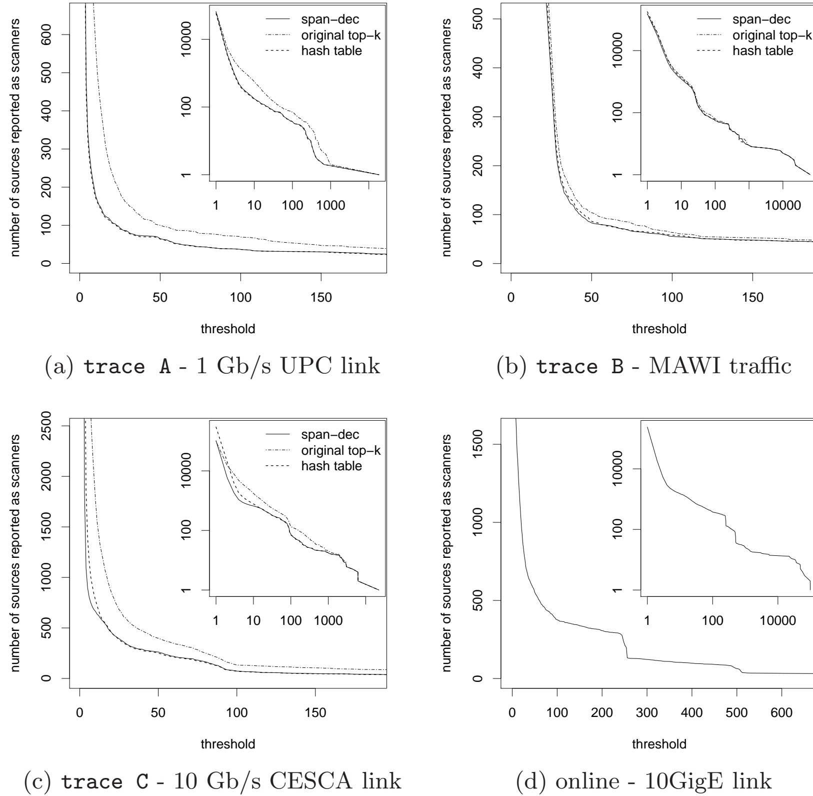
	`trace A`	`trace B`	`trace C`	`trace A0`
bf_syn size	256KB	256KB	1MB	64KB
bf_whitelist size	128KB	128KB	512KB	32KB
$span_{max}$	6	4	10	5

bf_syn using the unique number of $[A_{ip}, B_{ip}, B_{port}]$ tuples. The value of $span_{max}$ depends on the number of *Syn* packets concurrently sent by a source to distinct active destinations, which are not yet in the whitelist. We set this value according to 95th percentile of the traffic. For *topk* we arbitrarily set $elem_{max}$ to 10000 elements, unless otherwise noted. Resulting parameters are presented in Tab. 2. More details about the dimensioning procedure can be found in [10].

Detection threshold. To present the results for traces `A`, `B` and `C`, we follow the methodology used in [13]. Fig. 2 depicts the results when running the algorithm on our traces with the parameters described in Tab. 2. We plot the total number of sources reported as scanners as a function of the detection threshold. The threshold is the number of failed connections over which we classify a source as a scanner. The embedded plots show the whole range of data in a log-log scale, while the main plot presents only the part where the number of reported sources grows rapidly, in a linear scale. The “hash table” line presents the results obtained using hash tables to count distinct *Syn* and *SynAck* packets. In this scenario, all packets are counted with perfect accuracy. Results placed above this line indicate the presence of False Positives (FP), while those placed below the line imply False Negatives (FN). “Span-dec” line plots the results obtained when our counting structure was used. Both lines almost overlap indicating that our algorithm is close to an ideal tracking scheme using a hash table, but without its memory constraints. In particular, for high threshold values our algorithm features almost perfect performance. “Original top-k” shows the results obtained with the original *Stream-Summary* structure [9]. The large number of FP shows the necessity of supporting decrements in the counting structure.

Accuracy. The results in Fig. 2 were not enough to validate the actual accuracy of our algorithm. For this purpose, we used the ground truth `trace A0`, for which we knew the actual scanners and legitimate hosts. Our results show that, for thresholds higher than 20, the algorithm obtained perfect accuracy (i.e., 0 FP, 0 FN, and 100% detected scanners). More details about the accuracy of our algorithm and the impact of each configuration parameter are given in [10].

Filter performance. Tab. 3 presents the performance of the filters. The *Space usage* row shows the maximum space usage of each Bloom filter and (in brackets) the empirical collision probability. The probabilities are very small, even negligible. The *evictions* row shows the rate of traffic dropped by each filter (relative to the input packets of that filter). *Total packets evicted* gives the total ratio of handshake packets discarded by any of the two filters. Both filters together



(a) `trace A` - 1 Gb/s UPC link (b) `trace B` - MAWI traffic

(c) `trace C` - 10 Gb/s CESCA link (d) online - 10GigE link

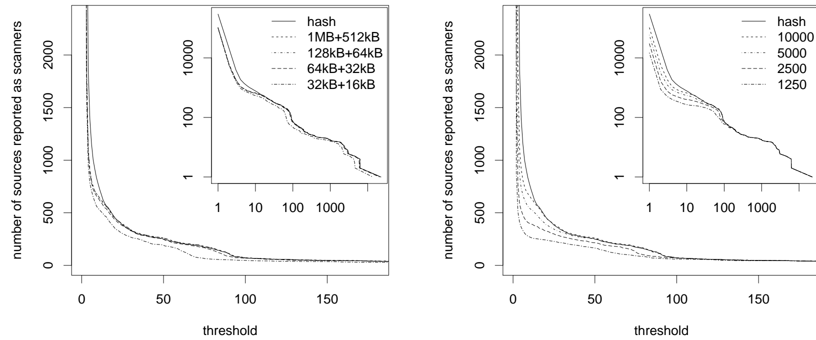
Fig. 2: Evaluation results on the traces - number of sources reported as scanners vs. detection threshold. Main graphs show a part of the data in a linear scale, embedded graphs show the whole range of data in a logarithmic scale.

drop about 85% of all handshake packets. Thus, only 15% of all *Syn*/*SynAck* packets result in increments or decrements in the counting structure. Given that the counting error depends directly on the number of introduced elements, with a smaller number of entries we achieve better accuracy with less space.

Memory size. Finally, we evaluated the impact of the memory size on the accuracy of the detection algorithm using `trace C`. First, we examined the impact of the size of the Bloom filters using 10000 entries in the *topk* structure. Results are presented in Fig. 3a. Filters below 96KB present FN due to collisions, as discussed in Sec. 2.1. With filters of 192KB (128KB+64KB) and a threshold above 100, the algorithm performs very close to the optimal. Using these filters, we examined the influence of the maximum number of elements ($elem_{max}$) in the *topk*. The results are presented in Fig. 3b. We can see that, for thresholds above

Table 3: Usage of the filters during the evaluation (evictions: *Syn* / *SynAck*)

	trace A	trace B	trace C
space usage: *bf_whitelist*	6.78% (6.59e-09)	1.90% (8.94e-13)	4.66% (4.77e-10)
space usage: *bf_syn*	13.27% (7.25e-07)	29.07% (1.75e-4)	11.02% (1.97e-07)
evictions: *bf_whitelist*	52.7% / 67.1%	24.7% / 76.2%	54.3% / 77.9%
evictions: *bf_syn*	61.2% / 65.0%	54.3% / 72.0%	55.4% / 64.2%
total packets evicted	84.3%	73.5%	84.4%



(a) filters (*bf_syn*+*bf_whitelist*) (b) max. number of elements in *topk*

Fig. 3: Impact of the memory size compared to an ideal scheme (`trace C`).

100, even with 2500 elements in the *topk* we still obtain very good accuracy. In our implementation, this configuration occupies only 417KB for a 10 GigE link.

Online deployment. In order to evaluate the real-time performance of the algorithm, we implemented it in the CoMo system [4] and deployed it on the 10GigE link from where `trace C` was collected. The hardware platform consisted of a PC with an Intel Xeon at 2.40GHz with two DAG 5.2SXA cards. A filter to discard non-*Syn*/*SynAck* packets was set in both cards. The filtering also can be done easily in software, since it requires only checking *Syn* and *Ack* flags in a TCP header. We ran the program for 100 min. (13-12-2010 at 10:50). The average traffic in the link was 5.4 Gb/s. The CPU load was about 5% during the whole experiment. For both filters, the maximum usage was 18.5% with a maximum collision probability of 7.31e-06. The threshold-alarm graph is presented in Fig. 2d.

4 Conclusions and Future Work

In this paper, we presented a practical approach to detect port scans in very high-speed links. The key idea behind our approach was to discard as much traffic as possible at early processing stages in order to reduce both the CPU and memory

requirements of our algorithm. We used two simple Bloom filters that maintain a whitelist of active destinations and efficiently track TCP handshakes, and combined them with an efficient top-k data structure to track failed connections. Both Bloom filters together can early discard about 85% of all handshake packets in our traces.

Our evaluation with four traces from different scenarios showed that our algorithm can achieve almost perfect accuracy with very little memory. We also deployed our algorithm in an operational 10GigE link and showed that it can work online. Also, we made a new dataset available to the research community, so that our results can be validated and compared with other solutions.

Although in the paper we focused only on TCP port scans, we are currently investigating how to extend the algorithm to detect UDP scans. A possible solution is to define which address blocks are behind the network to be protected. Another limitation of the algorithm is that it focuses on detecting top sources of port scans, and therefore it is not designed to reliably detect slow scans or more sophisticated attacks, like distributed scans.

References

1. UPC/CESCA traces: http://monitoring.ccaba.upc.edu/portscan/traces
2. MAWI Working Group Traffic Archive: http://mawi.wide.ad.jp/mawi/
3. http://www.bbc.co.uk/news/world-middle-east-11414483 (2010)
4. Barlet-Ros, P., et al.: Load shedding in network monitoring applications. In: Proc. of USENIX ATC (2007)
5. Brauckhoff, D., Tellenbach, B., Wagner, A., May, M., Lakhina, A.: Impact of packet sampling on anomaly detection metrics. In: Proc. of ACM SIGCOMM IMC (2006)
6. Jung, J., et al.: Fast portscan detection using sequential hypothesis testing. In 2004 IEEE Symposium on Security and Privacy
7. Mai, J., Sridharan, A., Chuah, C.N., Zang, H., Ye, T.: Impact of packet sampling on portscan detection. IEEE J. Select. Areas Commun. (2006)
8. Manerikar, N., Palpanas, T.: Frequent items in streaming data: An experimental evaluation of the state-of-the-art. Data & Knowledge Engineering (2009)
9. Metwally, A., Agrawal, D., Abbadi, A.E.: Efficient computation of frequent and top-k elements in data streams. In: Proc. of ICDT (2005)
10. Mikians, J., et al.: Span-Dec data structure in portscan detection. Technical report. http://monitoring.ccaba.upc.edu/portscan/portscan-report.pdf (2010)
11. Nam, S., Kim, H., Kim, H.: Detector SherLOCK: Enhancing TRW with Bloom filters under memory and performance constraints. Computer Networks (2008)
12. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. (1999)
13. Robertson, S., et al.: Surveillance detection in high bandwidth environments. In: Proc. of DARPA Information Survivability Conference and Exposition (2003)
14. Schechter, et al.: Fast detection of scanning worm infections. In: Proc. of RAID (2004)
15. Sridharan, A., Ye, T., Bhattacharyya, S.: Connectionless port scan detection on the backbone. In: Proc. of IPCCC (2006)
16. Weaver, N., Staniford, S., Paxson, V.: Very fast containment of scanning worms. In: Proc. of the 13th Conf. on USENIX Security Symposium (2004)