Practical Automated Detection of Stealthy Portscans

Stuart Staniford James A. Hoagland Joseph M. McAlerney

Silicon Defense
513 2nd Street
Eureka, CA 95501

Until some brilliant researcher comes up with a better technique, scan detection will boil down to testing for X events of interest across a Y-sized time window.
Stephen Northcutt [8]

Abstract

Portscan detectors in network intrusion detection products are easy to evade. They classify a portscan as more than N distinct probes within M seconds from a single source. This paper begins with an analysis of the scan detection problem, and then presents Spice (Stealthy Probing and Intrusion Correlation Engine), a portscan detector that is effective against stealthy scans yet operationally practical. Our design maintains records of event likelihood, from which we approximate the anomalousness of a given packet. We use simulated annealing to cluster anomalous packets together into portscans using heuristics developed from real scans. Packets are kept around longer if they are more anomalous. This should enable us to detect all the scans detected by current techniques, plus many stealthy scans, with manageable false positives. We also discuss detection of other activity such as stealthy worms, and DDOS control networks.

1 Portscanning

Portscanning is a common activity of considerable importance. It is often used by computer attackers to characterize hosts or networks which they are considering hostile activity against. Thus it is useful for system administrators and other network defenders to detect portscans as possible preliminaries to a more serious attack. It is also widely used by network defenders to understand and find vulnerabilities in their own networks. Thus it is of considerable interest to attackers to determine whether or not the defenders of a network are portscanning it regularly. However, defenders will not usually wish to hide their portscanning, while attackers will. For definiteness, in the remainder of this paper, we will speak of the attackers scanning the network, and the defenders trying to detect the scan.

There are several legal/ethical debates about portscanning which break out regularly on Internet mailing lists and newsgroups. One concerns whether portscanning of remote networks without permission from the owners is itself a legal and ethical activity. This is presently a grey area in most jurisdictions. However, our experience from following up on unsolicited remote portscans we detect in practice is that almost all of them turn out to have come from compromised hosts and thus are very likely to be hostile. So we think it reasonable to consider a portscan as at least potentially hostile, and to report it to the administrators of the remote network from whence it came.

However, this paper is focussed on the technical questions of how to detect portscans, which are independent of what significance one imbues them with, or how one chooses to respond to them. Also, we are focussed here on the problem of detecting a portscan via a network intrusion detection system (NIDS). We try to take into account some of the more obvious ways an attacker could use to avoid detection, but to remain with an approach that is practical to employ on busy networks. In the remainder of this section, we first define portscanning, give a variety of examples at some length, and discuss ways attackers can try to be stealthy. In the next section, we discuss a variety of prior work on portscan detection. Then we present the algorithms that we propose to use, and give some very preliminary data justifying our approach. Finally, we consider possible extensions to this work, along with other applications that might be considered. Throughout, we assume the reader is familiar with Internet protocols, with basic ideas about network intrusion detection and scanning, and with elementary probability theory, information theory, and linear algebra.

There are two general purposes that an attacker might have in conducting a portscan: a primary one, and a secondary one. The primary purpose is that of gathering information about the reachability and status of certain combinations of IP address and port (either TCP or UDP). (We do not directly discuss ICMP scans in this paper, but the ideas can be extended to that case in an obvious way.) The secondary purpose is to flood intrusion detection systems with alerts, with the intention of distracting the network defenders or preventing them from doing their jobs. In this paper, we will mainly be concerned with detecting information gathering portscans, since detecting flood portscans is easy. However, the possibility of being maliciously flooded with information will be an important consideration in our algorithm design.

We will use the term *scan footprint* for the set of port/IP combinations which the attacker is interested in characterizing. It is helpful to conceptually distinguish the footprint of the scan, from the *script* of the scan, which refers to the time sequence in which the attacker tries to explore the footprint. The footprint is independent of aspects of the script, such as how fast the scan is, whether it is randomized, etc. The footprint represents the attacker's information gathering requirements for her scan, and she designs a scan script that will meet those requirements, and perhaps other non-information-gathering requirements (such as not being detected by an NIDS).

The most common type of portscan footprint at present is a *horizontal scan*. By this, we mean that an attacker has an exploit for a particular service, and is interested in finding any hosts that expose that service. Thus she scans the port of interest on all IP addresses in some range of interest. Also at present, this is mainly being done sequentially on TCP port 53 (DNS). An example is shown in Figure 1. But other ports are common also, including 139 (NetBIOS file and print sharing), 98 (linuxconf), and 23 (telnet). The distribution of ports of interest changes over time as the popularity of different exploits grows and wanes in the attacker community.

Vertical scans are also seen. This is where an attacker scans some or all ports on a single host. Here the rationale is that the attacker is interested in this particular host, and wishes to characterize the services on it, perhaps with a view to find which exploit to attempt, or to find a suitable exploit via her network of contacts and resources. Part of an example vertical scan, produced using the popular nmap scan tool[4], is shown in Figure 2. In some cases, scans may only target a small range of ports. For example, a scan on just three ports is shown in Figure 3. This was an actual scan we detected, but the purpose of this scan, and the tool used to generate it, are currently unknown.

A scan may combine horizontal and vertical types into a *block scan* of numerous services on numerous hosts. More complicated geometries of what is to be scanned are possible in principal, though not seen much in practice.

Turning now to the individual scan probes, a number of types are known. Considering TCP first, perhaps the simplest type is for the scan tool to simply initiate the full three way handshake (nmap -sT for example). If the handshake succeeds, the port is open, whereas if it does not, the port is closed or perhaps filtered at some packet

filter or firewall device in front of the target address. This scan type is sometimes used by security consultants, but rarely by attackers. From an attacker's standpoint, the drawback to this method is that, since the *accept()* call on the socket at the server end has completed, the server application may generate a log entry for the connection (perhaps via TCP wrappers on Unix and similar systems). This leaves a trace of their activity unnecessarily.

The most popular TCP scan probe therefore is a syn-scan (nmap -sS). In this application, the scan tool generates just the initial syn packet. If this reaches an open port, the server will respond with a syn-ack packet. If the port is closed, the server will respond with a reset. If there is simply no response, this suggests that the port is firewalled between scanner and server. Modern scan tools can generate these syn packets to many hosts very rapidly, and then collect and collate the responses asynchronously as they arrive back at the scanning machine. Since the scan tool never completes the three way handshake, even if a syn-ack is returned, the host *accept()* socket call never completes successfully for that instance, and so no log of the scan is generated on the host machine (usually).

There are other scan probe possibilities. For example, syn-fin scanning involves sending packets with both syn and fin flags set. This is a combination that would never occur in normal TCP traffic. The attacker's hope in sending this is that an incorrectly implemented packet filter, which would not pass a bare syn packet, will pass the syn-fin packet. Fin scanning (just F flag alone) and XMAS scans (FPU) similarly hope to pass a firewall, and result in a reset from a closed port, and nothing from an open port.

Ack scanning involves sending an unsolicited packet with just the Ack flag set. It cannot distinguish open ports from closed, but can sometimes be used to map firewall rule sets (according to whether packets are dropped or result in resets).

Additionally, an important application of TCP portscanning is operating system (OS) identification. This relies on the fact that the TCP/IP RFCs do not specify how to handle illegal unanticipated flag combinations. Thus different implementations do it differently, and these idiosyncrasies can be used to determine what OS is in use at a particular IP address. For example, one popular tool, QueSO [14], sends seven packets:

0 SYN
1 SYN+ACK
2 FIN
3 FIN+ACK
4 SYN+FIN
5 PSH
6 SYN+1+2

(1 and 2 denote the reserved TCP flags.) All packets have a random sequence number and a 0 ack field. The replies are examined for their flag combination, ack, and similar, and this is compared to a table of popular OS's.

Current scanners can determine many kinds of operating systems - for example, nmap v2.53 can distinguish 468 different cases. On the face of it, this represents about 9 bits of information. However, there are probably only a couple of bits of true entropy per host because most hosts will have one of just a few popular operating systems.

UDP scanning is a little different from TCP scanning in that, if the port is open, the server will typically not respond to the probe (since the probe packet will not usually be a valid request in whatever protocol is being used over that port). However, if the port is closed, the server should respond with an ICMP message of Type 3 and Code 3 (Destination Unreachable because Port Unreachable). Thus UDP scanning involves listening for the ICMP responses from closed ports and then assuming that any ports that do not respond are open. This is sometimes slow (since hosts may rate limit how they send out the ICMP messages) and unreliable (since packets may be lost with no way to tell). UDP scans are much rarer in practice than TCP scans at present, but they certainly are possible and do occur.

For more detail on portscanning techniques, see [5, 6]. Up to date information can presently be found at [4].

From the standpoint of a network intrusion detection system, all the scan probe types which involve illegal flag combinations are extremely straightforward to detect. Rules which simply flag any packet with a non-standard combination of flags will detect all such scans with essentially no false positives. Thus, although these are often referred to as "stealth scans", they are not stealthy for our case. A network intrusion detection system is much more challenged by full connect scans, syn scans, and UDP scans where the individual packets could, on the face of it, be normal traffic. In the remainder of the paper, we shall concentrate on these cases.

It is helpful for us to be able to characterize how "big" a scan footprint is, or how big the portion of the footprint which falls into some particular organization's domain of IP space is. Clearly this tells us a lot about how difficult a scan is going to be to detect, and thus in measuring the efficiency of portscan detection, it's useful to parametrize against the size of the scan footprint. For example, a scan of every port on every host of a full Class C network involves the attacker checking 16,646,144 distinct host/port combinations. This is going to be hard to hide. By contrast, an attacker who only wants to know whether a single port is turned on, on a single host on our network, will find it much easier to evade detection.

The simplest method, then, of sizing a footprint is just to count the IP/port combinations the attacker needs to test. We call this the *total size* of the footprint. From the attacker's standpoint, this is often a good metric of how big her scan is, as it may be directly related to the num-

ber of bits of information about the target network that she obtains as a result of performing her scan. However, this is not always so - sometimes the attacker is attempting to obtain a number of bits of information from each port/IP combination. OS detection is the most obvious case in point. Thus it's useful then to talk about the *total information* of the footprint, as being the total number of bits of information the scan is intended to obtain.

However, there are alternative metrics that are useful from a detection standpoint. To motivate these, we consider that some pieces of the scan may be far easier for us to detect than others. Most obviously, if a scan probe finds an open port, it is much harder for us to determine that this was a scan than if it hits a closed port. This is because, other than some misconfiguration, normal traffic will not hit closed ports very often. Normal Internet users do not generally attempt to find services by making connections to the host/port in question to see whether the service is there. Rather, they rely on various forms of advertisements to tell them where services are (for example, links on web servers to other web servers, DNS information, information about where mail servers are that users enter into their computers manually). Thus connections to closed ports are inherently a lot more suspicious than connections to open ports (though by no means guaranteed to be intrusive). So another measure of the size of a scan footprint is the *closed size* of it - the number of distinct port/IP combinations the scan is targeting which are in fact closed at the time of scanning.

Indeed one approach to portscan detection is not to look at the scan packets (which we will call *forward scan detection*), but rather to look for packets that could be responses to portscan probes from closed server ports; TCP resets in the case of TCP scans, and ICMP port unreachable packets in the case of UDP scans. We refer to this as *backward scan detection*. The advantage of backward scan detection is that the packets are inherently more anomalous than the packets used in forward scan detection. The drawback is that we will miss portscans into empty IP addresses, which are particularly diagnostic. Forward and backward scan detection are both of value, and complement one another. In the remainder, we will generally use the language of forward scan detection for the sake of definiteness, but most of our techniques apply equally to both cases.

We present one final metric, which is the one we make most actual use of, and which is a generalization of the idea of the closed size of a scan. Suppose that the current probability distribution of normal traffic to ports and hosts on the protected network is known (in practice, it can be estimated from samples, but is not known perfectly). Then, when faced with any given scan probe for a given port/IP combination x, it is possible to determine the probability $P(x)$ that a normal traffic packet would be targeting this port. Then we can give a packet an *anomaly score* $A(x)$ as the negative log likelihood of this probe:

$$A(x) = -\log(P(x)) \tag{1}$$

Now the footprint of the scan is defined by a set X, of individual x. We can therefore define the total anomaly score of the scan as

$$A(X) = \sum_{x \in X} A(x). \tag{2}$$

Note that this is *not* usually the log likelihood of the overall scan, but it is a convenient measure of how easy it will be to detect this particular scan. The more unlikely the port/IP combinations the attacker needs information about in terms of our usual traffic, the more easily we should be able to detect her.

It's important to understand that the anomaly score of a scan is site dependent. If two sites have exactly the same configuration of hosts and services, and both sites are scanned in an identical way, the anomaly score of the scans may be quite different if the probability distribution of traffic is different. Even if the two usual traffic distributions are isomorphic, if the volume of normal traffic on one site is much greater than on the other site, then the scan will be more anomalous on the high traffic site.

We also note that we are making a simplification here. The probability distribution of traffic is time dependent. If the scan is spread out over time, then the anomaly scores for different probes are defined with respect to different probability distributions, which makes our definition of the total anomaly score for the scan footprint logically incoherent. We do not think this issue is of practical importance at present, and so we ignore it.

We now turn to looking at what an attacker can do with her script to make it hard for us to detect her investigating her chosen footprint. A variety of techniques are available to her.

Changes of scan order. Most scans in the wild at present move through IP addresses sequentially, going from lower to higher. However, if this assumption is used by defenders in detection, it is straightforward for attackers to change it. Randomizing the order in which IP addresses and ports are searched can easily be done. nmap is currently capable of randomizing the addresses it uses within blocks of *2048* hosts. Also, if an attacker suspects that a particular detection algorithm is in use, the scan order can be constructed maliciously to put that detection algorithm into its worst case performance.

Slowing down. By slowing down the scan, an attacker can make it more difficult to detect. This easily defeats current naive scan detectors by simply extending the scan so that successive probes appear out of the detection window. It forces any detector to keep more state, and pick the scan pattern out of far more normal traffic. Thus detection becomes more difficult. The price the attacker pays is that it takes more time for her to obtain the information

she needs. Thus it may be useful to characterize scans by the average information rate the scan is achieving (how many bits of information per second it discovers on average).

Randomizing inter-probe timing. Deterministic delays between probes can help some detection algorithms. Therefore, it is of value for an attacker to insert random delays into the probes. An exponential waiting distribution would be a natural thing to try to introduce noise into the process, but power law distributions with long tails could also be used, since it is known that network traffic distributions often have features of self-organized criticality [9].

Randomizing non-essential fields. Fields such as sequence number, ack number, IP id, and source port in the scan packets are often hard coded with fixed values in current scans. Alternatively, they may be generated with some simple deterministic algorithm. This makes detection easy, and so attackers are likely to randomize them in future.

Affecting the source address. The source address is more difficult for an attacker to affect, and so is a key piece of information for scan detection. In the simplest case, the attacker does need to use a real source address, since she needs to see the packets that servers generate in response to the scan in order to know what ports are actually open.

An exception to this is if an attacker is able to monitor a network close to the target network (perhaps the ISP of the target network). In that case, the attacker is free to forge the source addresses randomly, and rely on monitoring to see the response packets. This idea has been implemented in at least one tool - Icmpenum [15]. This idea will often not be practical as the attacker may not be able to compromise the ISP, or if she can, the fast switched networks there may not be amenable to network monitoring.

However, as a diversionary tactic, it is certainly feasible for an attacker on a network that does not perform egress filtering to create additional probes with forged addresses. Nmap has a mode to do just this, invoked by command line option '-D'. This makes it more likely the scan will be detected, but harder to determine what response to make.

Distributed scanning. An attacker who can launch her scan from a number of different real IP addresses can investigate different parts of the footprint from different places. This complicates the detection task. In the extreme case, large networks of agents similar to those used for distributed denial of service attacks could be used for portscanning. As of early 2000, such tools were under development, but not in widespread use. It is reasonable to assume that portscanning may evolve in this direction.

We should assume that all these tactics will be used by attackers. Some are in use already. It may take several years for tools with all these features to be in widespread use, but it will certainly happen eventually. And sophisticated attackers with large budgets could develop tools with these features in several months' effort (and may have done so already).

It is worth noting also that some common events look like hostile portscans but are not. A variety of network computer games will, on startup, contact a range of different servers very rapidly (often using UDP). The scans often use a default port, but with variations depending on the individual server. An example from the popular game "Half Life" is shown in Figure 4.

Also, web pages will sometimes contain elements located on several different servers (ads, scripts, and graphical elements may be in a variety of different places). When the browser loads the page, it will issue a quick burst of DNS lookups and port 80 connections as it assembles all the resources it needs to render the page. This may trigger present day portscan detectors.

2 Prior work in scan detection

To our surprise, there seems to have been very little work on the problem of efficiently and reliably detecting portscans. Given all the research in intrusion detection over the last decade and a half, and the enormous practical importance of this problem, it is striking how little attention it has received. A number of research IDS systems, data mining studies, etc., must have detected portscans, but how they did so is not generally published or commented upon. Commercial systems generally use the naive algorithm to the best of our knowledge. However, they may often be unwilling to reveal their algorithm choices. We survey the few relevant systems here.

2.1 NSM

The Network Security Monitor (NSM)[7] was the first NIDS, and also the first NIDS to detect scanning. It had rules to detect any source IP address connecting to more than *15* other source IP addresses (presumably within some time window, but this is not clearly specified in the paper). Thus it pioneered the algorithm that has been used by most systems ever since.

2.2 GrIDS

The Graph Based Intrusion Detection System (GrIDS) prototype was built by UC Davis[16, 2] (the team involved two of the present authors). It was intended to detect rapid automated hostility of various kinds, including portscans. It was the first system to attempt to do this on a large scale using hierarchical processing.

GrIDS detected portscanning by building graphs of activity in which the nodes represented hosts, and the edges represented some network traffic between hosts. Thus a scan probe could be represented as an edge between the

scanning host and the server being scanned. GrIDS assembled these edges into graphs based on the fact that the edges shared at least one node, and on other user definable rules. Thus scans in which all the probes had the same source IP could be detected. In practice, the rules were usually conditioned on time so that only scans that occurred fairly rapidly were detected. This was not a limitation in principle, however, whereas the restriction to same source IP of probes was.

GrIDS had a complex design which allowed it to propagate information about graph edges up a hierarchy of processing engines which viewed the network on larger and larger scales. This enabled it to detect even very sparse random scans as long as they were rapid and used the same source IP.

GrIDS had no notion of anomaly or probability for packets, so it would always be limited in its ability to handle stealthy scans. Additionally, the prototype implementation was in Perl and was quite slow for modern networks. Nonetheless it was used in practice for a number of months on a network of about a hundred hosts and was quite useful there.

2.3 Snort portscan preprocessor

Snort [11, 12] is an open source lightweight network intrusion detection system based on libpcap [1]. It can produce real-time alerts as well as packet logs in a variety of formats. Snort has a flexible rules language to describe what alerts should be alerted, logged, or passed. Different members of the Snort community provide rules that can be used for a particular installation and sites can write their own rules. The detection engine uses a modular plugin architecture, which allows developers to extend Snort and users to choose the functionality required to meet their needs.

The portscan detection functionality in Snort is made possible by a preprocessor plugin. The Snort portscan detector attempts to look for X TCP or UDP packets sent to any number of host/port combinations from a single source host in Y seconds, where X and Y are user defined values. Additionally, the portscan detector looks for single TCP packets that are not used in normal TCP operations. Such packets will have odd combinations of TCP flags set, or no flags set at all.

Upon arrival, a packet's structure is checked for soundness. The packet is then tested to see if it is part of a scan currently in progress. This is achieved by comparing the packet type and source address to those of scans currently being investigated. If it is not part of a current scan, it becomes the starting node of a new scan. Otherwise, the matching scan's packet count is incremented, and a check is made to determine whether the threshold of X packets sent in Y seconds was exceeded. If so, the scan is reported. The scan will also be reported, regardless of the threshold being broken, if the packet contained an abnormal TCP flag combination.

The current version of the Snort portscan detector has a couple notable shortcomings that can easily be used to evade portscan detection. First, it is unable to detect scans originating from multiple hosts. Also, the threshold is determined by a static combination of user specified numbers. The threshold is usually set high enough to allow for only a bearable amount of portscan false positives. As a result, it is very easy to avoid detection by increasing the time between sending scan probes.

2.4 Emerald

The EMERALD system[10] from SRI International has also been used to detect portscanning, and uses a different algorithm than the usual one. EMERALD can regard each source IP address communicating with the monitored network as a *subject*. It constructs statistical profiles for subjects, and matches a short term weighted profile of subject behavior to a long term weighted profile. When the short term profile goes far enough into the tails of the distribution for the long term profile, EMERALD views it as suspicious. One of the aspects of subject behavior can be the volume of particular kinds of network traffic generated. This can be used to detect portscanning as a sudden increase in the volume of syn packets, for example, from a particular source IP.

This approach has some limitations. It is not capable of detecting slow stealthy scans, since those will not create the kind of sharp volume increase that EMERALD looks for. It also cannot easily correlate distributed source scans. And finally, it is not clear how EMERALD would interpret scanning from IPs that have never been seen before and which have no profile.

3 Spice detection approach

So how might we detect a slow randomized scan which is buried in days, weeks, or months of normal traffic? If we only use a short detection window, we will miss slow scans. However, if we try to use a long detection window, we face searching through massive amounts of normal traffic looking for patterns. It's generally completely infeasible to save all network traffic for any length of time since there is so much of it.

The key insight that we invoke is that the attacker is trying to gather information which she does not already know, and she is trying to find out this information in some systematic way, rather than simply approaching the target site as any normal user would. (Of course a good attacker will have used less noticeable reconnaissance first, but if she is portscanning, it implies a desire to know about ports that may or may not be open.) Therefore, at least some of the portscan is likely to be highly anomalous traffic relative to the usual traffic distribution. If the packet has unusual features (i.e., is a "crafted packet") this will be

still more true. Thus our approach is to save information about packets to be searched later based on how anomalous the packet is. Thus a TCP syn to port 98 (linuxconf) on a Windows host will be saved for much longer than one to port 80 (http) on a known web server. This allows us to accumulate such rare events over a longer period of time.

We then try to group the saved packets together into activities that are similar, using simulated annealing with a variety of heuristics discussed later. Packets that fall into a sizeable group are also saved longer, thus meaning that a stealthy portscan will be saved, grouped, and noticed while normal traffic on the site will timeout and be lost from state quickly.

Architecturally, Spice has two kinds of components: an anomaly sensor and a correlator. The sensor (Section 3.1) monitors the network and assigns an anomaly score to each event. Those events that are sufficiently anomalous are passed, along with their anomaly scores, to the Spice correlator (Section 3.2). This correlator groups events together and reports scans. Section 3.3 briefly discusses theoretical limits on what Spice can detect. We discuss how the anomaly sensor, Spade, was implemented in Section 3.4 and how we implemented the correlator in Section 3.5.

3.1 Anomalous event assessment

As discussed in Section 1, we assess the anomalousness of an event based on the probability that a normal event would look like the event. This is based on packet header fields such as source IP, destination IP, source port, destination port, protocol (TCP or UDP), and protocol flags. Some combination of these should go into the characterization of the packet for these purposes. The optimal way to do this in general could be the subject of years of research and may vary with the monitored network. We describe two general approaches. In either case, in order to assess the anomalousness of events, the sensor will need to maintain probability tables of feature instances and multi-dimensional tables of conditional probabilities observed. We discuss how we implemented this efficiently for Spade in Section 3.4.2.

The first approach is to directly maintain the joint probability of a certain set of features. That is, directly measure things like *P(destination port,destination IP,source IP,source port)*. This has the advantage of simplicity. However, if there are more normal combinations of this than are seen regularly on the network, then the result can be noisy and not reliable. Also, maintaining all the different combinations of values may be resource intensive. Generally, employing as few features as needed to characterize the packet's probability should make this more tractable.

The second approach is to construct an estimate based upon more limited probabilities and conditional probabilities such as $P(destinationport)$, $P(sourceport|destinationport)$, and $P(destinationIP|sourceIP, sourceport)$ using a construction known as a belief network, or Bayes network [13]. A Bayes network is a diagram describing how variables in a system of variables are related. For example, it will describe whether two variables are independent or conditionally independent. If variables are independent then there is no need to measure their joint probabilities. A Bayes network that we provide in Spade is shown in Figure 5. The Bayes network allows us to estimate the joint probability distribution while only measuring the conditional probability distribution of pairs and triples of packet header fields (which is a more tractable thing to do). To derive the full joint probability of a packet, the product of the probabilities of each feature in the network, given that the parent features(s) (if any) have the values that they do. Thus while the conditional probability of source port given destination port is needed, the conditional probability of destination IP given destination port is not.

The way in which we assessed full and conditional independence in designing our network is by making entropy measurements on live traffic. We compute the amount of entropy in different fields of the packet header, and then compute the mutual information between various fields of interest. This allows us to assess quantitatively which fields are related to which others. With real network traffic, we were not able to establish the total independence of fields, but it might be close enough. Given the way in which we use the result, it should not be overly sensitive to this effect. Figure 6 shows the relationship between source IP, source port, destination IP, and destination port in real network traffic. This is based on 3 weeks of observation of 1,258,251 TCP syn packets on the network of a small company. Note that once the source IP and source port are known, not much remains unexplained about the destination IP. Providing the destination port as well explains not much more (just 0.166 bits). So, the Bayes net shown asserts that destination IP is independent of destination port, given source IP and source port.

3.2 Portscan correlation in Spice

Events which have an anomaly score greater than a certain threshold at the sensor will be sent to the correlator. There, they are assembled into groups as described in this section.

The challenge that we face is that we wish to consider a number of heuristics in determining exactly what to group, and we don't know a priori which heuristics will be helpful in grouping any particular scan. For example, in one scan, the fact that the ID fields in the IP packets are all the same will be very helpful, while in another scan, the fact that source port and IP address proceed in lockstep is what will be helpful.

The other challenge is that we do not know in what order and how quickly the events to be grouped will arrive - the ordering and timing may even be malicious. Thus

deterministic algorithms can easily be led astray. In this setting, we turn to statistical physics algorithms for inspiration (though we will use them in a loose and creative way).

3.2.1 Correlation graph

The metaphor that motivates our approach is this: the events (packets) to be correlated are like atoms living in space. Each heuristic is expressed as a bonding energy between the atoms. Then we create *bonds* between those events where the attraction is strongest. These are described in a graph, where the events are nodes and the bonds are undirected edges. We refer to this as the *bond graph.* Roughly speaking, two events will be bonded if there is a strong connection between the events. Each bond has a certain strength associated with it. As a constraint, all events are connected in a single graph.

Groups of related events are represented in the graph by subgraphs in which each connection is above a certain strength. The events in such subgraphs describe an entire group. Thus the groups are the connected subgraphs left when bonds weaker than some threshold are deleted.

An example correlation graph is shown in Figure 7.

3.2.2 Evaluation function

The strength of connections between events is evaluated pairwise. The form of the evaluation function is:

$$f(e_1, e_2) = c_1 h_1(e_1, e_2) + c_2 h_2(e_1, e_2) + \ldots + c_k h_k(e_1, e_2)$$

where e_1 and e_2 are the events whose connection is being evaluated, $c_1 \ldots c_k$ are constants, and $h_1 \ldots h_k$ are heuristic evaluation functions. The heuristic evaluation functions capture knowledge of how events are connected in scans. The methods by which a heuristic may operate are arbitrary. (We believe that a set of simple heuristics can capture most portscans seen in the wild today and that it is feasible to capture the stealthier portscans that are likely to be more common in the future with somewhat more sophisticated heuristics.) So as to prevent any heuristic from exerting undue influence over the evaluation function, all heuristic evaluation functions are required to produce results within the continuous range [0,1], where 0 indicates the heuristic finds no connection between the events and 1 indicates the strongest possible connection between the events. Initial heuristics are likely to include:

- *Feature equality heuristics.* Is the source IP address the same between the events? The destination port? How about the destination network? If so, 1. Else, 0.
- *Feature proximity heuristics.* How close are the times of the events? How about the destination IP? Or the destination port? The closer, the closer to 1 the result would be. If they are too far apart, the result is 0.
- *Feature separation heuristics.* This heuristic attempts to recognize gaps in a feature's value between events. This can be done in a primitive way using just the two immediate events being evaluated. It might recognize well-known separations between events, e.g., 1 hour between events or a step of 1 in destination IP. Considering the bond partners of events allows more sophisticated analysis. It can recognize that a certain feature has the same gap between two events as between another two events. An efficient implementation might find the bond partner of one of the events with closest to the same gap as the reference events and evaluate the heuristic based on this closeness.
- *Feature covariance heuristics.* Recognizing event features that vary together can be implemented to a limited extent by looking at just two events. It can be noted, for instance, that source port is rising at the same rate as destination IP (e.g., (destip1 - destip2)/(srcport1 - srcport2)=1) and that would result in a high value. Considering the events bonded to some event allows more sophisticated heuristics. If e_1 and e_2 have a rate between two features that is the same as the rate between e_2 and some bond partner of e_2, there is a connection and the heuristic would yield higher results. This would be useful for destination IP to time based rates.

The constants $c_1 \ldots c_k$ are expected to be within a small factor of each other for active heuristics.

3.2.3 Adding Events

When a new event is presented to the correlator, it needs to be added to the graph somehow. There may be a large number (perhaps thousands) of events in the graph so a new event cannot be tested for connections with every event. We use the technique of simulated annealing [13], assigning some number (perhaps 4) of bonding partners randomly initially (though we may wish to make one of these the last event added as a possible optimization).

Simulated annealing is a technique that was originally developed in statistical physics for finding the global energetic minima of physical systems. However, it has since been used in a variety of applications in AI and other fields for problems associated with finding the configuration that globally maximizes some evaluation function in the presence of local maxima. The name and algorithm derive from an explicit analogy with annealing which is the process of gradually cooling a liquid till it freezes. It can be viewed as a variation on a hill-climbing algorithm. In the hill climbing algorithm, all successor states (neighbors) of a given state are evaluated and the one that evaluates the highest is chosen as the new reference point. This continues until all successors are worse than the present. This termination condition is a problem if there are local maxima in the state space. Simulated annealing overcomes

this problem by sometimes making moves to states that seem worse.

At each step in simulated annealing, a possible successor is chosen randomly among the successors of the current state. If the successor state evaluates better than the current state, it is made the next current state. If it is worse, then a move is made to it only with certain probability. This probability decreases with the degree to which the possible successor state is worse than the current state and is regulated by a cooling schedule. The cooling schedule controls the degree to which negative moves are allowed. The cooling schedule is defined by a function that starts off high (at 1) and decreases to 0 by the time at which the global maxima is expected to be found. When it is at 0 it is essentially hill-climbing.

As we apply it to the process of finding bond partners for a new event, the states are events in the bond graph and the evaluation function is the one introduced in the previous section, applied between the new event and the other event. Successors of a state are the existing bond partners of the current event. The exception to this is if a successor already has a bond with a new event, in which case its bond partners are considered successors but it is not. We denote the schedule function by *sched*, where *sched(t)* is its value at time *t*. This becomes 0 in some finite time. We will illustrate the process with the following algorithm, where *maxe* is the highest possible value of the evaluation function:

```
add_event(new)
  for i= 1 to 4
    current= randomly chosen event in the graph
    t= 1
    while (sched(t) > 0)
      neighbor-set= neighbors(current,new)
      next= randomly chosen event from
                              neighbor-set
      change= e(new,next) - e(new,current)
      if (change > 0 or (rand(0,1) <
         e^(change/(sched(t)*maxe))))
         current= next
         t++
   create a new bond between
              current and new

neighbors(refevent,avoid)
  set n to {}
  for each event e in a bond from refevent
    if there is not an bond between
                              e and avoid
      n= n + e
    else
      n= n + neighbors(e,avoid)
  return n
```

Figure 8 shows some example steps in the annealing process. Initially, a new event e_{100} is given a bonding partner e_{17} at random; their link evaluates a strength of 1.7. e_{22} is chosen as the next event since it is a neighbor of e_{17}. The link between e_{100} and e_{22} evaluates to 4.0, better than the link with e_{17}, so it becomes the current event. e_5 is chosen at random among e_{22}'s bonding partners as the next possible successor. Although it evaluates lower (2.1), it is selected anyway by chance. e_{74} is not as lucky; it evaluates lower than e_5 but it is not selected and e_5 remains the current event. e_{22} is chosen as the possible successor to e_5 and is selected due to its higher evaluation. e_{50} is chosen among e_{22}'s bonding partners and it becomes the new current event since it evaluates higher.

When other correlators provide groups of events for consideration, they are added individually to the graph using the process described above. There is no attempt to maintain global consistency as to the disposition of event groups. This would be complex in general and this way allows individual correlating sites to decide on their own parameters and heuristics.

Weak bonds in the graph that are not necessary to maintain graph connectedness would be discarded as a low priority background operation. This helps keep the graph tidy.

3.2.4 Timing out events

All events time out. How long they are around depends on their anomaly score and the other events in the group of which they are a part. An event's individual lifetime is proportional to its anomaly score, so the most anomalous events are retained longer. This is added to the time of the event to find its scheduled timeout. However, events in a group inherit the latest timeout of any event in the group. This keeps events around while they are part of an active group and rewards being bonded to long lived events.

If, after removing events that have timed out, the graph is separated, then an operation must take place to reconnect the graph. The way to determine what disconnected subgraphs are created by removing events is to check for paths between the events that neighbor the removed events. If there is a path, they are in the same subgraph. Provided there is more than one subgraph, reconnection proceeds as follows. For each pair of subgraphs, the strongest pairwise bond between events in the two subgraphs is added to a list sorted by decreasing strength. This produces a list of the best way to merge any two subgraphs. To determine which of these to employ to merge all the subgraphs together, repeatedly removed the top of the list. If the events are still in different subgraphs, a bond is added between the events. This continues until all the subgraphs have been merged. We believe that this procedure is a good tradeoff between performance and bond strength optimization. We expect that in the typical case, there would be only a small number of disconnected subgraphs formed because a well connected event seems less

likely to time out.

3.2.5 Sharing and alerting scans

The anomaly score for a group of events is the sum of the anomaly scores for the events in a group. This is used when deciding when to share or alert groups of events. If a certain threshold is exceeded, the group of events is shared with other correlators. If a separate (presumedly higher) threshold is exceeded, then a group of events is sent as an alert to a user.

3.3 Limits on detection

In this section we take an intuitive look at the theoretical bounds on what Spice can detect. Formalizing this is left as an exercise for the reader or for the authors at a later date.

Spice is the component that classifies a set of events as a portscan. However, the only events it considers are those that the anomaly sensor produces. So we first consider the anomaly sensor. The anomaly sensor passes along statistically anomalous packets to Spice. As argued in Section 1, the vast majority of scan packets will be statistically anomalous and thus will be passed to Spice. The ones that will not be are the ones that happen to look like normal traffic. The importance of their omission from a portscan report depends on the application for which Spice is being used. However, there is likely to be a pattern apparent in the scan, so that the missing events, if any, can be inferred.

An event fails to be detected as part of a portscan if it has not been included in with a group that gets reported as a portscan. As described in Section 3.2.4, a group times when each of the component events reach the end of their individual lifetimes. Thus, a group must be kept fresh in order to be around to include any later portscan events. This means that much depends on the constant that is used as a multiplier to calculate the individual lifetime of an event. If the attacker allows a time greater than the individual lifetime of an event to pass before sending the next event in the scan, then there is no way for Spice to detect the scan. (Given a reasonably long lifetime factor though, it is unlikely that attacker would be patient enough for the results of the portscan. The results could even be out of date by the time the scan completes.)

Due to the nondeterministic approach used in adding events to the bond graph, there is a small chance that a new event will not be grouped with another event in the same portscan that is already present in the bond graph, even if the bond score would have been high enough. Such an possibility (the occurrence of which is controllable to an extent by the cooling function used in simulated annealing) would delay the inclusion of an event in a portscan group and thus could be a factor in a missed detect.

For a given bond evaluation function, not all pairs of events in a portscan are necessarily within a given grouping threshold. This could occur even if the bond evaluation function is constructed well enough to include all scan event within the grouping threshold. Thus, the ordering of the events in the scan could be important. For example, the attacker could attempt to schedule the events such that a small group of portscan events would time out before the necessary events occur to enable it to be included in a large enough group to be reported. (Even given a relatively patient attacker, it is not known whether this is something that can be practically achieved. The geometry of the scan footprint being targeted by attacker may also put limits on this.)

To improve the reporting of events in a scan, there is pressure to increase event lifetimes and to lower the grouping threshold used. The tradeoff with this is that non-scan events could also be included in portscan reports. Lowering the reporting threshold and using a less specific (weaker) bond evaluation function might also improve the reporting of scan events, but would increase the chance that non-scan traffic is reported as a scan. Fine-tuning these trade-offs is most likely to be determined empirically and would depend on the goals of the organization using Spice.

3.4 Spade

We have an implementation of the Spice anomaly detector publicly released under GNU GPL. It is called SPADE (Statistical Packet Anomaly Detection Engine) and can be downloaded from http://www.silicondefense.com/-software/spice/. It is a Snort preprocessor plugin, which gives us the benefit of using Snort's input/output facilities such as receiving packets already parsed into a data structure. This is where we maintain the probability tables that are used to assign an anomaly score. In its present form, Spade only looks at TCP syn packets since this where the interesting truly stealthy scans are now (by design though, it can easily handle other packet types).

The portscan correlation will run in a separate process, possibly on a remote machine (see Section 3.5). The communication between the anomaly detector and the correlator is via sockets and consists of the anomaly detector passing along details of anomalous events along with their anomaly scores. We think that having separate processes and communicating via sockets makes sense for a couple of reasons. First, this way Snort does not take too long in processing any packet, which might otherwise lead to dropping packets. The correlating process has a little more liberty to do extra computations with the anomalous events. Also, if other correlators want to communicate anomalous events that they have found, then they can send it to the correlating process and not to Snort.

3.4.1 Spade features

Spade has a number of features that can be enabled and configured through the Snort configuration file. It offers the user four alternatives for assessing the likelihood of packets. One is the Bayes network depicted in Figure 5. The other three are direct joint probability measurements: $P(sourceIP, sourceport, destinationIP, destinationport)$, $P(sourceIP, destinationIP, destinationport)$, and $P(destinationIP, destinationport)$. The user may also elect to have Spade only monitor packet going into certain networks. This allows Spade to focus its assessment on the traffic of interest, removing the noise of outgoing traffic (which typically has a much larger range of possible addresses and ports).

Since Spade maintains state over a period of time, it provides checkpointing and recovery facilities. Spade starts up recovering its state from a specified file and periodically (and on signals and Snort exit) stores its state in a designated file.

The anomalous event reporting threshold is an important parameter in a Spade installation. Unfortunately it is also one whose ideal value varies from site to site depending on the characteristics of the network. This could also vary over time. If the threshold is too high, interesting events may be missed. If it is too low, the use may be flooded with events, most of which are not interesting. To allow the user to get Spade running well "out of the box" with minimal threshold adjustment, three capabilities are provided by Spade to automatically adjust the threshold to observed network traffic. These aim to meet a specified target rate (in term of packet count or in terms of a fraction of traffic).

Spade also provides two modes unrelated to its primary purpose of reporting anomalous events. One is a survey mode in which statistics about the distribution of anomaly scores recently observed are appended to a file periodically, thus producing a table of this information. The other is the capability to report on certain known feature statistics such as entropy and conditional probabilities. It is this functionality that produced the measurements shown in Figure 6.

3.4.2 Maintaining probabilities in Spade

Depending on the probability mode, Spade needs to maintain certain joint probabilities for packet features. (Conditional probabilities needed for the Bayes network calculation can be derived from unconditional probabilities.) The most efficient way to do this in a real time system is to maintain a count of features in observed events. Conceptually, for a feature A, whose probabilities are needed, there would be a table with the different possible values of A and a count of their occurrences. To determine the probability of a particular value of A, its count is divided by the total number of events recorded in the table. If we need to know the probability of the joint occurrence of $a \in A$ and $b \in B$, then we need a two dimensional table, where the entry for a and b records a count of their joint occurrence. In general a k-dimensional table is used to record the joint occurrence of k feature-values.

Now comes the question of how to efficiently represent these tables. The nature of network traffic influences this. Certain feature values may be much more likely than others (e.g., destination port 80 may be much more likely than destination port 5037). In fact, the observed values may be sparse compared to the total range of possible values for a feature, so an array representation (while it would be efficient for lookups) would be too expensive in terms of memory usage. Hash tables can be similarly inefficient and it would be difficult to find a good general hash function that is not biased with the data for all cases. For the conditional probability tables, we would require hash tables of pointers to other hash tables. This would certainly make for much waste of space for rows in the table that were almost empty.

It is also important to have a data structure which will perform tolerably well even when the sensor is seeing a flood scan designed deliberately to fill up the data structure with all possible cases that could occur in the table. This rules out linked lists and similar structures.

We take a general approach. We have decided on a custom data structure and algorithm based on a balanced binary search tree. Our aim is a solution which generally provides very fast access for the common cases (main servers and most popular ports), but can handle very large growth in the number of entries in the structure while still maintaining tolerable performance and being space efficient.

Let us first introduce the data structure as it would be used for a single dimension. A tree is maintained that stores all the values observed. These values are stored in leaf nodes along with a count of the number of instances observed. As is standard in binary search trees, these nodes are kept in order from left to right. Interior nodes record the largest value on the left side of the node. This serves as an indication whether to go left or right to look for (or insert) a leaf node of a particular value. Interior nodes also maintain the sum of the counts of the leaf nodes beneath the node. Consider as an example the tree in Figure 9, depicting counts of destination ports.

It is this sum that is the focus of balancing in our efforts to maintain the tree. We feel the counts serve as a predictor of future accesses. Specifically, we wish for the left and right child nodes of all interior nodes to have as close to the same count/sum as possible. The result of this is a tendency to push leaf nodes with higher counts higher in the tree, since they have more weight for the balancing than other nodes. This results in more efficient access for this common case.

The need to rebalance a subtree is checked periodically.

This period is in terms of the number of count increments in the subtree. To support this, a wait count is maintained on each interior node. This count is decremented with each increment in the subtree. When an interior node is created and after a rebalance check, the wait count is set to the greater of 10 or the minimum number of insertions that would be needed to unbalance the subtree. Any interior nodes whose children were changed in the process of rebalancing are also checked for rebalancing. Also to avoid frequent rebalancing, no effort is made to rebalance a subtree unless one side is more than twice the size of the other side.

To rebalance a subtree, left and right rotations (see [3]) are performed. If the left has a higher count/sum than the right, then a right rotation is done; otherwise a left rotation is done. In addition to right and left rotations, more general relocation of subtrees from right to left and left to right are performed if needed for rebalancing. This is repeated for the node in that position in the tree until a further rotate would bring the tree more out of balance than current or until no further rotations can be done since a leaf node and the bottom of the tree is encountered.

As an example of rebalancing, consider the balanced binary search tree in Figure 10, which is the tree shown in Figure 9 after 4 additional port 80 observations. Notice that the children of the root are unbalanced. After a left rotate, this would be rebalanced to the tree shown in Figure 11.

To use this structure in two dimensions, the type of trees described would also be used for a second dimension and would be anchored off the leaf nodes in the first dimension. This is extended in a straightforward manner for more than two dimensions.

The characteristics of a network will change over time and the most attention should be paid to recent characteristics. Furthermore, if we were to store artifacts of every access indefinitely, this would lead to a large amount of memory use and large data structures. The approach taken in Spade to this is to de-emphasize past observations periodically with respect to new observations. It would be too inefficient to scale all current counts down by certain amount with each new event (and ultimately too inflationary to increasingly emphasize new events, besides which it would not eliminate old one-time events). Instead, we take the optimization of only doing the de-emphasis on occasion. For example, every hour we might multiply all counts by 99.5%, discarding occurrences with too low a result (say below 0.25). Thus an observation that occurred once (and was given an initial weight of 1) would only have a weight of 0.886 after 24 hours.

3.4.3 Spade results

Though our results are preliminary, Spade seems stable and efficient. We have had it running for over 3 months on a client's Internet connection without problems. Using the 3 week data set (see Section 3.1), we measured that Spade processed the file in about 2 minutes, including producing reports. This is an average of about 86 microseconds per packet. Memory use was between 2 MB and 42 MB depending on the probability mode employed.

In using it for our commercial monitoring with a threshold setting that typically produces about 300 alerts per day, Spade has noticed (at least) most of the events in every TCP syn portscan that we would have noticed otherwise. In addition, there are many slow or small scans we have detected though the Spade alerts that we would not have noticed otherwise.

As a step in assessing the effectiveness of Spade in detecting actual portscans, we identified 28 horizontal scans (consisting of 1245 packets) and 4 nmap network scans (107026 packets) in the 3 week data set. (There may have been scans we did not find in that data set.) We then compared this against the alerts produced when Spade was run in different configurations. We present some of our results here.

We calculate two indicators, which we term *efficiency* and *effectiveness*. Efficiency is the ratio of the number of true positives to all positives. For these experiments, it is the number of alerts that had been labelled as part of one of the scans divided by the number of alerts produced. The bigger this number is, the less noise the correlator will have to deal with. Effectiveness is the ratio of true positives to all trues. This is how well Spade detected scan packets. For us, this is the number of alerts that had been labelled as part of one of the scans divided by the number of labelled events.

There is a tradeoff between these indicators. Generally, if you want increased effectiveness (that is, you want to catch more of the scans), the lower your efficiency will be (that is, you will have more noise). This is illustrated in Table 1. This shows the results of running Spade over all the packets in the data set using the two feature joint probability mode with static threshold settings of 12, 13, 14, and 15. The higher the cutoff, the higher the efficiency but the lower the effectiveness.

However, using Spade's homenet option improves both. The homenet was set to cover the IP addresses of the monitored network. This leaves 1,010,909 packets whose destination is in the home network. The two joint probability mode is used again with static threshold settings of 9, 11, and 13. The results are depicted in Table 2. For any of these settings, the efficiency is above 85% and the effectiveness is above 99.7%. The reason for the improvement seems to be that outgoing traffic could not be adequately sampled in terms of destination port and IP combination due to the wide variety of destinations.

Table 3 depicts some results of comparing the different probability modes available in Spade (see Section 3.4.1). We used Spade's adapt3 mode, which very slowly adjusts the threshold to meet a target rate. As configured, ev-

ery hour the rate is adjusted by averaging the last 168 measurements of what threshold would have caused 0.3% of packets to be sent as alerts. The homenet option was again employed. The joint probability mode with 2 features clearly does the best here, detecting 99.75% of those scan packets. To truly have a fair comparison between the modes, we should compare the results when they are in the configuration that works best for them (e.g., at their best reporting threshold).

These preliminary results serve to support our belief that Spade can detect portscan packets well, but that it may take some amount of configuration work to find an optimal configuration. Note also that a low efficiency rate might be acceptable when Spade is being used to feed the correlation engine, part of whose task it is to weed out non-scan events.

3.5 Correlator Implementation

We have completed a detailed design of the correlator and have nearly completed our initial implementation. We are using a multi-threaded approach. The threads currently implemented are: a thread to receive anomalous event reports into a queue, threads to add events from the queue to the bond graph and to report scans, a thread to clean up the graph (removing weak links), and a thread to time out and remove events from the graph. Also anticipated are a checkpointing thread, a thread to respond to queries about correlator state, and a thread to receive commands to adjust operational parameters dynamically. Certain read and write mutual exclusion locks need to be maintained for data shared between threads (e.g., the bond graph, the queue, and the timeout data structure).

The bond graph has a pretty straightforward representation for traversing the graph. Two operations that a graph representation is not good for are discussed in the following section.

3.5.1 Timeout data structure

The representation of the bond graph is not suited for maintaining event timeouts or choosing random events from the graph. Thus we maintain a separate timeout structure for these operations. A simple example of this is depicted in Figure 12.

At a conceptual level, the timeout structure maintains a record of events and when they will time out. As an optimization, it actually records the scheduled timeout – the time it was to time out in the last instance it was checked. This may be different than its timeout value at a given moment since the timeout might be delayed due to new events that are added to its group. When the clock reaches the value of an entry in the timeout tree, each event associated with that entry is checked. If the timeout is still current, then it is deleted from the timeout structure and from the bond graph. Otherwise it is reinserted into the structure with its new scheduled timeout value.

At an implementation level, we use a hash table to maintain the list of times at which timeouts are scheduled to occur and for which events. A simple hash function is employed, $h(t) = t \bmod N$, where N is the size of the hash table array. When multiple timeout times hash to the same bucket in the hash table array, a linked list of times (and the associated events) is used for the bucket. This linked list is kept sorted by increasing timeout time.

We also use this timeout structure in selecting random events. Our strategy for picking events with uniform probability is to conceptually order all the events in the hash table, to choose an order position by choosing a integer with equal probability among the positions. The corresponding event would then be retrieved. We order the events as follows: first by hashtable array position (smallest to largest), then by timeout time, and finally by event position in the list of events at the time. (It does not matter that the event list is in arbitrary order or that the timeout values may not be current; we only care that the order is well defined at any given time.)

Standard hash tables are not efficient for selecting events in this manner (retrieving an event in an arbitrary position), so we augment the hash table with an array of size $N-1$ maintaining certain counts of events in a range of hash table array slots. As is often done with the heap data structure [3], we view this count array as a complete binary tree. The tree root is node 1, the left of node i is $2*i$, and the right of node i is $2*i+1$. For simplicity in discussion, we assume than N is a power of 2. The tree is a full one of height $log_2(N)$ in this case. The $N/2$ leaves of this tree each (conceptually) have adjacent left and right nodes in the timeout hash table array. The count maintained on each node in tree is the number of events below its left node. For example, node 2 in the figure stores the number of events in the first hash bucket and node 1 contains the number of events below node 2. This allows a particular event position to be located by walking down the tree from the root. To further increase selection efficiency, we maintain a count of events at a given timeout (not depicted in the figure), so that events in certain times can be skipped entirely.

3.5.2 Correlator results

As of the time of this writing, we had just begun formal experimental testing of the correlator. The results from informal testing are promising. In several scenarios using real data with introduced scans, we detected and grouped the events of randomized horizontal portscans with inter-probe gaps as high as 90 minutes, even using several source IP addresses. There were few false positives or extra events grouped in. In addition to formal experiments, we have plans to validate SPICE in operational environments.

4 Future Work and Extensions

In this work, we have sketched the design for Spice, which we hope will be a viable method of picking stealthy portscans out of background traffic. We also described the implementation of the Spice anomaly sensor, Spade and the Spice correlator. We will conduct formal experiments with Spade and Spice to measure its detection performance. Operational validation will also be done. These may suggest ways to tune Spade and Spice.

We note also that if this tool is successful, it should also be very useful for detecting the spread of worms and the use of distributed denial of service networks. Like portscans, those applications involve large numbers of connections or packets with similar structures which will typically be quite anomalous relative to regular traffic. The only difference from detecting portscans will be some change in the heuristics. Thus Spice could be a tool capable of detecting such misuses of the network *without* first reverse engineering the particular worm or DDOS tool in use.

We also note that Spice lends itself in a natural way to distributed or hierarchical use. We could share events upwards or sideways only if they were particularly anomalous (more so than required just to correlate them locally). This would allow a set of Spice correlators to collectively detect and characterize very sparsely distributed network misuse across a number of autonomous networks. Hence, it might be possible for collaborating sites to compare strange events such as Figure 3, and determine whether they genuinely are isolated, or whether they are part of a larger pattern at present unseen.

5 About the authors

Dr. Stuart Staniford is President of Silicon Defense, an Intrusion Detection Research and Monitoring company in Eureka, CA. He received a Ph.D. in Physics from UC Davis in 1993, with a dissertation concerning the application of statistical physics techniques to particle physics. He received an MS in Computer Science from UC Davis in 1995 with a thesis about tracing intruders over the Internet. He worked under Karl Levitt. He then worked as a Researcher at UC Davis, leading the team that developed the GrIDS prototype intrusion detection system, which was the first IDS that attempted to do wide area detection of scans and worms. He was the founding chair of the Common Intrusion Detection Framework (CIDF) working group, which developed a data format for research IDS systems to share data. This led to the start of the IETF working group IDWG to develop an Internet standard for intrusion detection alerting, which Dr. Staniford now co-chairs.

Dr. Staniford left UC Davis to work full time for Silicon Defense, his own company, in mid 1999. There he works on ways for intrusion detection systems to share and correlate data, methods for traceback of intruders, improving the operational practice of intrusion detection, and promoting IDS standards. He is a member of the board of the Common Vulnerabilities and Exposures Project, and of the program committee for the Recent Advances in Intrusion Detection (RAID) workshop series. He strongly believes in the importance of bringing the research and operational intrusion detection communities together, and so is a frequent contributor on internet mailing lists about intrusion detection systems and incident analysis and handling.

Dr. James Hoagland is an Associate Researcher at Silicon Defense. His current work includes research on stealthy scan detection (part of the Multi-Community Cyber Defense project sponsored by US DARPA), development of the SnortSnarf IDS alert navigator, and intrusion monitoring. In the UC Davis Computer Science department, he received his BS in 1993, MS in 1996, and Ph.D. in 2000. His Ph.D. dissertation was on specifying and implementing security policies under the supervision of Profs. Raju Pandey and Karl Levitt. He was a Research Assistant in the UCD Computer Security Research Lab for six years, where he conducted research in security policies, intrusion detection (notably being an original GrIDS designer and implementer), audit log visualization and analysis, and network security. In summer 1997, he was a Graduate Technical Intern under the supervision of Dr. Baiju Patel at the Internet Security group of the Intel Architecture Labs.

Joseph McAlerney is a programmer and security analyst at Silicon Defense. He graduated from Humboldt State University in December of 1999 with a BS degree in Computer Information Systems. He has developed API's for the Intrusion Detection Inter-component Adaptive Negotiation (IDIAN) project. Joe maintains an Intrusion Detection Exchange Message Format (IDEMF) library called libidmef, as well as the IDMEF plugin for the Snort IDS. Both are implementations based on efforts made by the IETF IDWG working group. He also operates as a Snort consultant and second line Snort Support engineer at Silicon Defense.

6 Acknowledgements

This work was supported under DARPA contract #F30602-99-C-0181. We thank DARPA for their ongoing support of our research, and intrusion detection research in general. This paper was helped by discussions with some of our collaborators at Boeing (Randy Smith and Dan Schnackenberg), UC Davis (Karl Levitt, Jeff Rowe, and Dave Klotz), and NAI Labs (Dan Sterne, Kelly Djandahari, and Roshan Thomas). Dave Farrel and Raymond Parks at Sandia National Labs provided helpful ideas about the attacker's view of portscanning. The idea to use Bayes networks in this way came to us follow-

ing discussions with Al Valdes at SRI of his quite different use of Bayes networks in [17]. We also thank Marty Roesch, Patrick Mullen, and the rest of the Snort community for making available a viable open-source IDS; that was important in the development of this research. Finally, thanks to Steve Northcutt for the quote given at the start of this paper, which partially inspired this work. We aren't brilliant yet, but we aspire to be.

All opinions in the paper are those of the authors alone.

References

[1] http://www.tcpdump.org/.

[2] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, D. Zerkle. "The Design of GrIDS: A Graph-Based Intrusion Detection System." in: *U.C. Davis Computer Science Department Technical Report CSE-99-2*, 1999.

[3] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. MIT Press. Cambridge, MA. 1990.

[4] Fyodor. http://www.insecure.org/nmap/.

[5] Fyodor. The Art of Scanning, Phrack 51 www.phrack.com.

[6] Fyodor. Remote OS detection via TCP/IP Stack Fingerprinting, Phrack 54 www.phrack.com

[7] L.T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A network security monitor," in: *Proc., 1990 Symposium on Research in Security and Privacy*, pp. 296-304, Oakland, CA, May 1990.

[8] S. Northcutt, S. Network Intrusion Detection: An Analyst's Handbook. New Riders, Indianapolis, 1999. p. 125.

[9] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling, *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[10] P. Porras and A. Valdes, Live Traffic Analysis of TCP/IP Gateways. 1998 Internet Society Symposium on Network and Distributed System Security. San Diego, March 1998.

[11] M. Roesch. "Snort - Lightweight Intrusion Detection for Networks," in: *Proceedings of the 1999 USENIX LISA conference. November 1999.*

[12] M. Roesch. http://www.snort.org/.

[13] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hill. Upper Saddle River, NJ. 1995.

[14] Savage. http://www.apostols.org/projectz/queso/.

[15] Simple Nomad. http://razor.bindview.com/tools/-desc/icmpenum_readme.html.

[16] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, D. Zerkle, "GrIDS – A Graph-Based Intrusion Detection System for Large Networks". in: *The 19th National Information Systems Security Conference.*

[17] A. Valdes and K. Skinner. http://www.sdl.sri.com/-emerald/adaptbn-paper/adaptbn.html

threshold:		12.0	13.0	14.0	15.0
# of alerts		191718	168606	150082	136554
# events detected	horiz.	1195	1160	1107	984
	Nmap	106910	106904	106764	106492
	total	108105	108064	107871	107476
efficiency		0.5639	0.6409	0.7187	0.7871
effectiveness		0.9984	0.9980	0.9963	0.9926

Table 1: Spade results on the 3 week data set with threshold settings of 12, 13, 14, and 15 and with the 2 feature joint probability mode.

threshold:		9.0	11.0	13.0
# of alerts		127225	119234	114400
# events detected	horiz.	1205	1195	1143
	Nmap	106940	106912	106896
	total	108145	108107	108039
efficiency		0.8500	0.9067	0.9444
effectiveness		0.9988	0.9984	0.9978

Table 2: Spade results on the portion of the 3 week data set that has destination internal to the monitored network. Threshold settings of 9, 11, and 13 and the 2 feature joint probability mode were used.

probability mode:		joint-2	joint-3	joint-4	Bayes
# of alerts		116453	87889	478069	402596
# events detected	horiz.	1191	527	413	244
	Nmap	106815	44294	68290	79830
	total	108006	44821	68703	80074
efficiency		0.9275	0.5100	0.1437	0.1989
effectiveness		0.9975	0.4139	0.6345	0.7395

Table 3: Spade results for the different probability modes on the portion of the 3 week data set that has destination internal to the monitored network. Spade adapt3 mode was use with a target alert rate to 0.3% of packets.

```
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.193:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.194:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.195:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.196:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.197:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.198:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.199:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.200:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.201:53 SYNFIN
Apr 1 19:02:12 666.66.666.66:53 -> 111.11.11.202:53 SYNFIN
```

Figure 1: A fragment of a Snort portscan preprocessor log for a sequential DNS scan.

```
Apr 1 18:36:01 666.66.666.66:1093 -> 111.11.11.49:21 SYN
Apr 1 18:36:01 666.66.666.66:1094 -> 111.11.11.49:22 SYN
Apr 1 18:36:01 666.66.666.66:1095 -> 111.11.11.49:23 SYN
Apr 1 18:36:01 666.66.666.66:1096 -> 111.11.11.49:25 SYN
Apr 1 18:36:01 666.66.666.66:1097 -> 111.11.11.49:42 SYN
Apr 1 18:36:02 666.66.666.66:1116 -> 111.11.11.49:8010 SYN
Apr 1 18:36:02 666.66.666.66:1117 -> 111.11.11.49:8080 SYN
Apr 1 18:36:02 666.66.666.66:1100 -> 111.11.11.49:79 SYN
Apr 1 18:36:02 666.66.666.66:1102 -> 111.11.11.49:110 SYN
Apr 1 18:36:02 666.66.666.66:1101 -> 111.11.11.49:80 SYN
Apr 1 18:36:02 666.66.666.66:1104 -> 111.11.11.49:119 SYN
Apr 1 18:36:02 666.66.666.66:1103 -> 111.11.11.49:111 SYN
```

Figure 2: A fragment of a Snort portscan preprocessor log for a vertical portscan.

```
Apr 1 11:18:56 666.66.666.66:2419 -> 111.11.11.47:80 SYN
Apr 1 11:18:56 666.66.666.66:2420 -> 111.11.11.47:80 NOACK
Apr 1 11:19:00 666.66.666.66:2423 -> 111.11.11.47:80 SYN
Apr 1 11:19:00 666.66.666.66:2427 -> 111.11.11.47:80 SYN
Apr 1 11:19:31 666.66.666.66:2434 -> 111.11.11.47:37 SYN
Apr 1 11:19:31 666.66.666.66:2434 -> 111.11.11.47:37 NOACK
Apr 1 11:19:34 666.66.666.66:2435 -> 111.11.11.47:37 SYN
Apr 1 11:19:34 666.66.666.66:2435 -> 111.11.11.47:37 NOACK
Apr 1 11:19:37 666.66.666.66:2436 -> 111.11.11.47:37 SYN
Apr 1 11:19:38 666.66.666.66:2437 -> 111.11.11.47:13 SYN
Apr 1 11:19:38 666.66.666.66:2437 -> 111.11.11.47:13 NOACK
Apr 1 11:19:41 666.66.666.66:2438 -> 111.11.11.47:13 SYN
Apr 1 11:19:44 666.66.666.66:2439 -> 111.11.11.47:13 SYN
```

Figure 3: Portscan scanning 3 ports. This is the whole log.

```
Apr 1 21:16:21 111.11.11.197:4344 -> 23.222.22.222:27015 UDP
Apr 1 21:16:21 111.11.11.197:4345 -> 32.233.33.233:27015 UDP
Apr 1 21:16:21 111.11.11.197:4346 -> 34.244.44.244:27015 UDP
Apr 1 21:16:21 111.11.11.197:4242 -> 43.250.55.250:27016 UDP
Apr 1 21:16:21 111.11.11.197:4320 -> 45.100.66.100:27015 UDP
Apr 1 21:16:21 111.11.11.197:4329 -> 54.120.77.120:27015 UDP
Apr 1 21:16:21 111.11.11.197:4347 -> 56.180.88.180:27015 UDP
Apr 1 21:16:21 111.11.11.197:4354 -> 65.190.15.190:27015 UDP
Apr 1 21:16:21 111.11.11.197:4311 -> 67.200.55.200:27015 UDP
Apr 1 21:16:21 111.11.11.197:4205 -> 76.202.13.202:27015 UDP
Apr 1 21:16:21 111.11.11.197:4350 -> 78.195.13.195:27015 UDP
Apr 1 21:16:21 111.11.11.197:4355 -> 87.199.85.199:27015 UDP
Apr 1 21:16:21 111.11.11.197:4313 -> 89.190.95.160:27015 UDP
Apr 1 21:16:21 111.11.11.197:4356 -> 98.248.15.230:27015 UDP
Apr 1 21:16:21 111.11.11.197:4325 -> 90.123.16.157:27015 UDP
```

Figure 4: A fragment of a Snort portscan preprocessor log of a Half life scan.

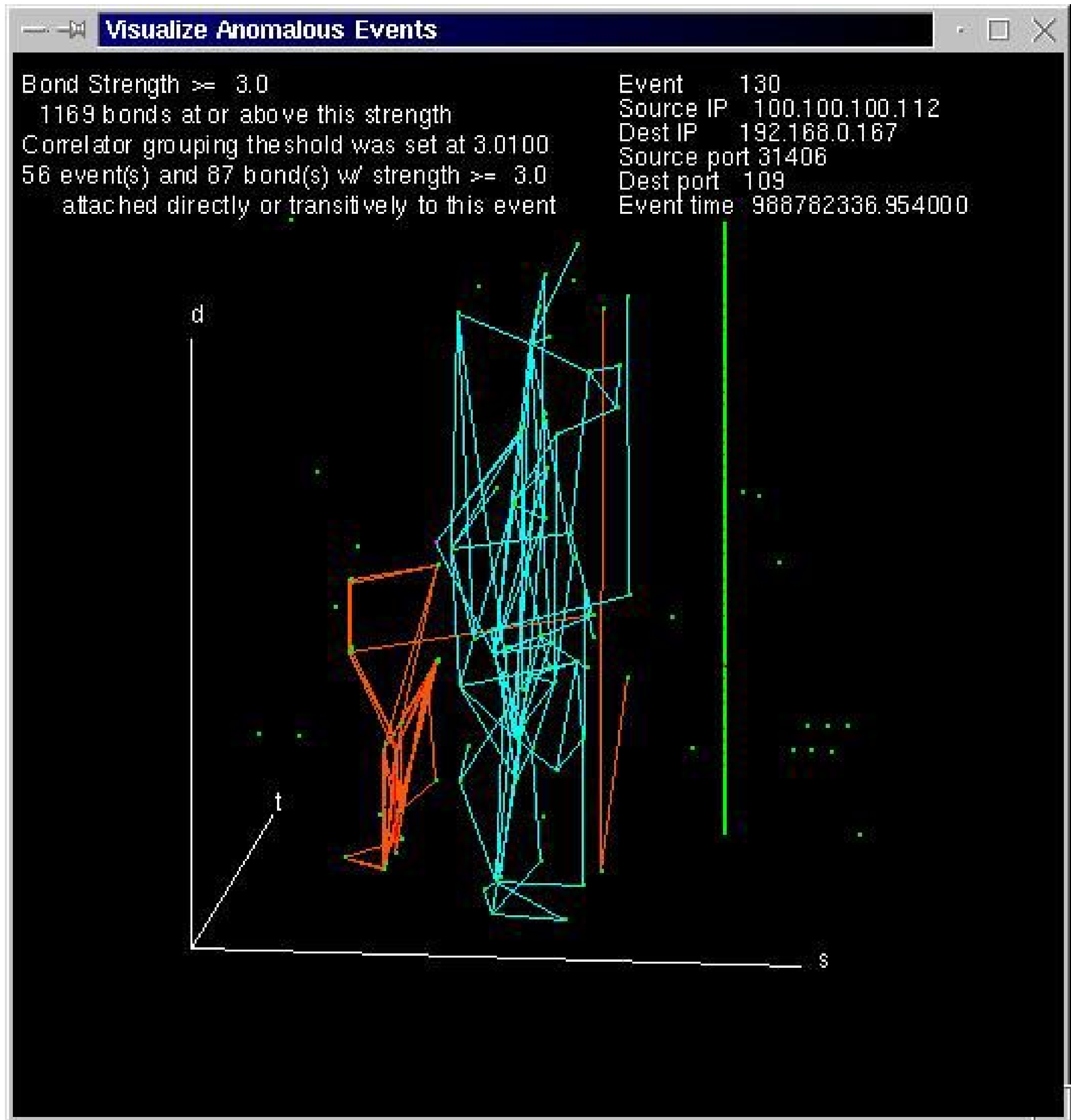

Figure 7: An example correlation graph. This is a screen shot of a tool to visualize a snapshot of a Spice correlator graph. It is a plot of events and the edges (bonds) between those that are above a threshold. The axes are source IP (s), destination IP (d), and time (t). The highlighted edges are part of the same group, which is a stealthy, distributed-source, portscan. The closely spaced series of green dots on the right side are an unstealthy horizontal portscan.

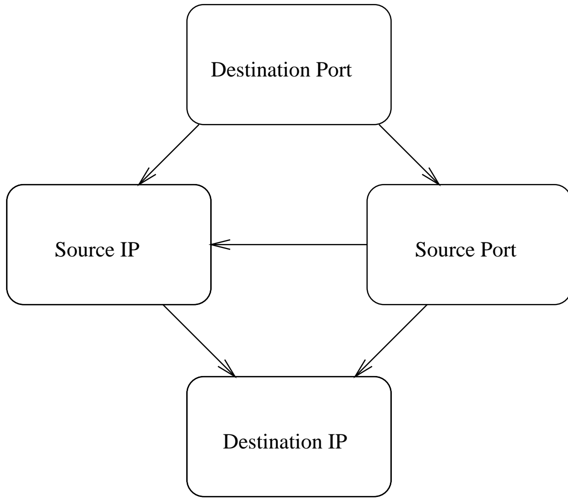

Figure 5: The Bayes network provided in Spade. Arrows indicate that one feature influences the other. For example, source IP and source port influence the destination IP.

$H(dip) = 4.602$
$H(dip|dport) = 2.876$
$H(dip|sip) = 1.616$
$H(dip|sport) = 2.750$
$H(dip|sip, dport) = 1.447$
$H(dip|sip, sport) = 0.530$
$H(dip|sport, dport) = 1.467$
$H(dip|sip, sport, dport) = 0.364$

$H(sip) = 7.980$
$H(sip|dip) = 4.995$
$H(sip|dport) = 6.048$
$H(sip|sport) = 4.225$
$H(sip|dip, sport) = 1.814$
$H(sip|dip, dport) = 4.619$
$H(sip|sport, dport) = 2.718$
$H(sip|dip, sport, dport) = 1.615$

$H(dport) = 3.118$
$H(dport|dip) = 1.393$
$H(dport|sip) = 1.186$
$H(dport|sport) = 1.737$
$H(dport|dip, sport) = 0.263$
$H(dport|sip, dip) = 1.018$
$H(dport|sip, sport) = 0.230$
$H(dport|sip, dip, sport) = 0.064$

$H(sport) = 13.938$
$H(sport|dip) = 12.278$
$H(sport|dport) = 12.557$
$H(sport|sip) = 10.183$
$H(sport|sip, dip) = 9.097$
$H(sport|sip, dport) = 9.227$
$H(sport|dip, dport) = 11.148$
$H(sport|sip, dip, dport) = 8.144$

Figure 6: Observed entropy amounts for source IP and port and destination IP and port among TCP syn packets. All numbers are bits of entropy. The conditional entropies are the amount of entropy that remain in a feature when the feature conditioned on are known.

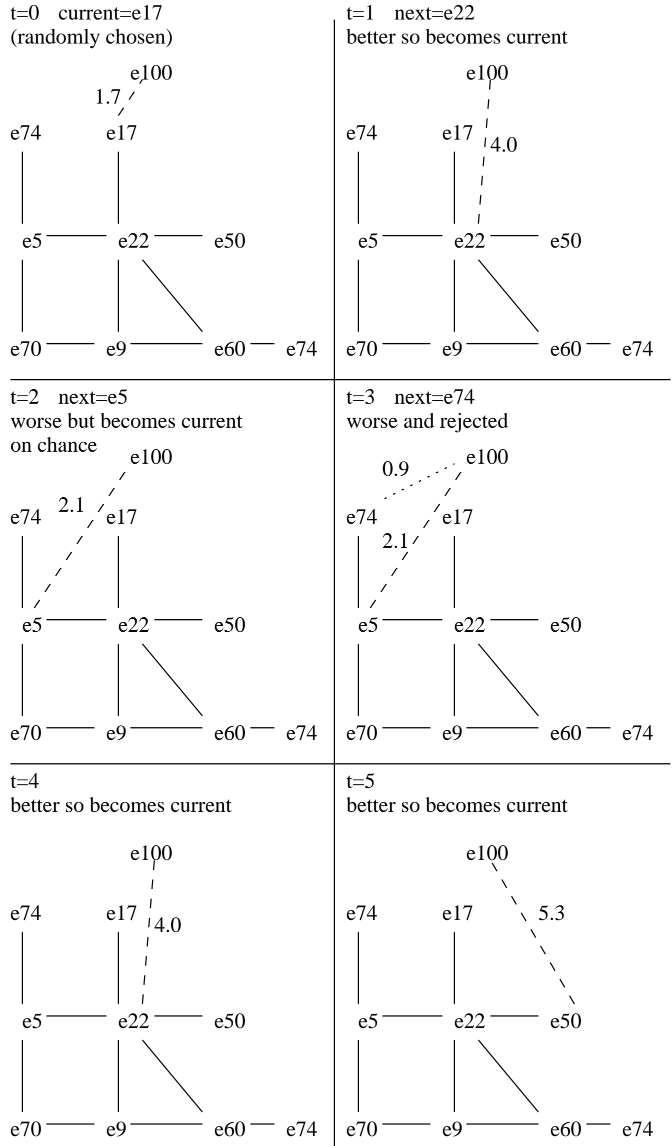

Figure 8: Some simulated annealing steps to add e100 to the correlation graph.

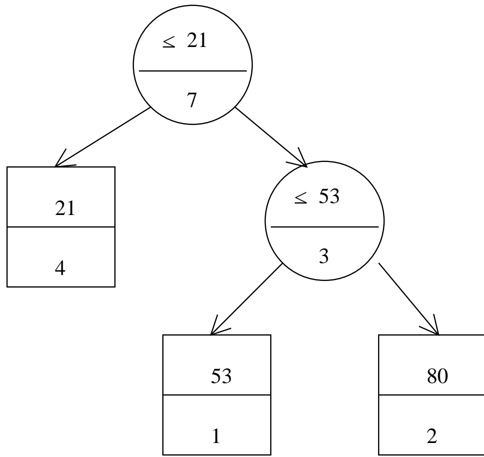

Figure 9: Balanced Binary Search Tree for counts of destination ports. The upper number in leaf nodes are the port number represented and the lower number is a count of instances. In interior nodes, the upper number is the indication of the position of value beneath and the lower number is the sum of instances counted below.

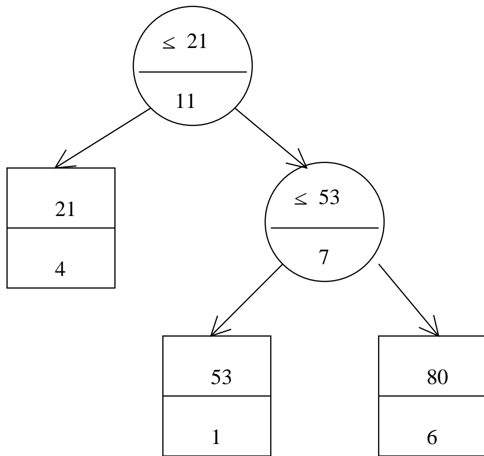

Figure 10: Balanced Binary Search Tree after 4 more port 80 observations.

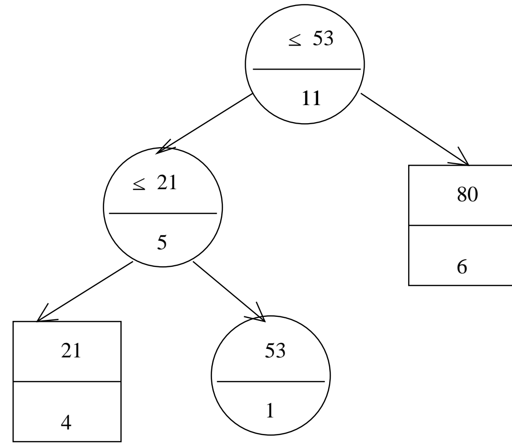

Figure 11: Rebalanced Balanced Binary Search Tree after a left rotate at the root.

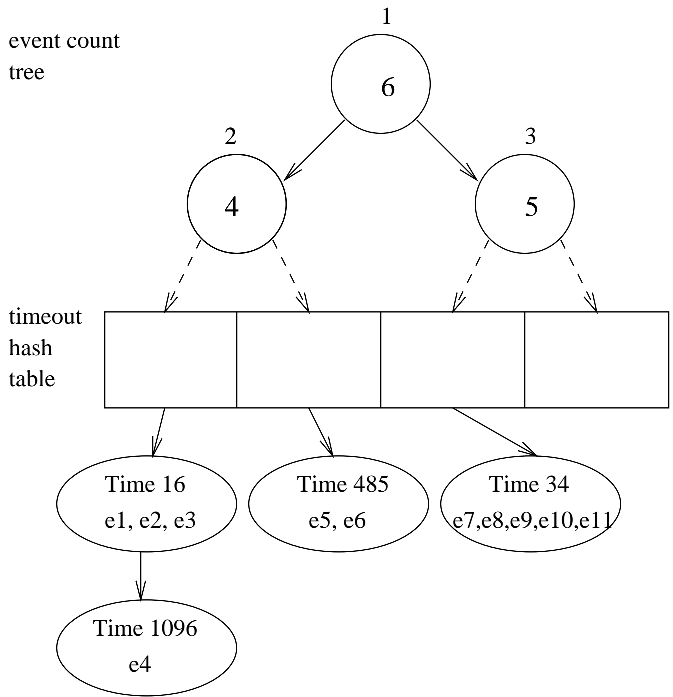

Figure 12: Timeout structure example. The lower part is the timeout hash table of size N=4. The upper part is the event count array depicted as a tree. Dashed lines show how these relate. The value on the event count tree nodes is the number of events beneath its left side. Event numbers illustrate the implicit ordering of events in the hash table.