CHAPTER

# Network Security Algorithms

# 4

From denial-of-service to Smurf attacks, hackers that perpetrate exploits have captured both the imagination of the public and the ire of victims. There is some reason for indignation and ire. A survey by the Computer Security Institute placed the cost of computer intrusions at an average of $970,000 per company in 2000.

Thus there is a growing market for *intrusion detection*, a field that consists of detecting and reacting to attacks. According to IDC, the intrusion-detection market grew from $20 million to $100 million between 1997 and 1999 and is expected to reach $518 million by 2005.

Yet the capabilities of current intrusion detection systems are widely accepted as inadequate, particularly in the context of growing threats and capabilities. Two key problems with current systems are that they are slow and that they have a high false-positive rate. As a result of these deficiencies, intrusion detection serves primarily as a monitoring and audit function rather than as a real-time component of a protection architecture on par with firewalls and encryption.

However, many vendors are working to introduce *real-time* intrusion detection systems. If intrusion detection systems can work in real time with only a small fraction of false positives, they can actually be used to *respond* to attacks by either deflecting the attack or tracing the perpetrators.

Intrusion detection systems (IDSs) have been studied in many forms since Denning's classic statistical analysis of host intrusions. Today, IDS techniques are usually classified as either *signature detection* or *anomaly detection*. Signature detection is based on matching events to the signatures of known attacks.

In contrast, anomaly detection, based on statistical or learning theory techniques, identifies aberrant events, whether known to be malicious or not. As a result, anomaly detection can potentially detect new types of attacks that signature-based systems will miss. Unfortunately, anomaly detection systems are prone to falsely identifying events as malicious. Thus this chapter does *not* address anomaly-based methods.

Meanwhile signature-based systems are highly popular due to their relatively simple implementation and their ability to detect commonly used attack tools.

The lightweight detection system Snort is one of the more popular examples because of its free availability and efficiency.

Given the growing importance of real-time intrusion detection, intrusion detection furnishes a rich source of packet patterns that can benefit from network algorithmics. Thus this chapter samples three important subtasks that arise in the context of intrusion detection. The first is an *analysis* subtask, string matching, which is a key bottleneck in popular signature-based systems such as Snort. The second is a *response* subtask, traceback, which is of growing importance given the ability of intruders to use forged source addresses. The third is an *analysis* subtask to detect the onset of a new worm (e.g., Code Red) without prior knowledge.

These three subtasks only scratch the surface of a vast area that needs to be explored. They were chosen to provide an indication of the richness of the problem space and to outline some potentially powerful tools, such as Bloom filters and Aho–Corasick trees, that may be useful in more general contexts. Worm detection was also chosen to showcase how mechanisms can be combined in powerful ways.

This chapter is organized as follows. The first few sections explore solutions to the important problem of searching for suspicious strings in packet payloads. Current implementations of intrusion detection systems such as Snort (www.snort.org) do multiple passes through the packet to search for each string. Section 4.1.1 describes the Aho–Corasick algorithm for searching for multiple strings in one pass using a trie with backpointers. Section 4.1.2 describes a generalization of the classical Boyer–Moore algorithm, which can sometimes act faster by skipping more bits in a packet.

Section 4.2 shows how to approach an even harder problem—searching for *approximate* string matches. The section introduces two powerful ideas: min-wise hashing and random projections. This section suggests that even complex tasks such as approximate string matching can plausibly be implemented at wire speeds.

Section 4.3 marks a transition to the problem of responding to an attack, by introducing the IP traceback problem. It also presents a seminal solution using probabilistic packet marking. Section 4.4 offers a second solution, which uses packet logs and no packet modifications; the logs are implemented efficiently using an important technique called a *Bloom filter*. While these traceback solutions are unlikely to become deployed when compared to more recent standards, they introduce a significant problem and invoke important techniques that could be useful in other contexts.

Section 4.5 explains how algorithmic techniques can be used to extract automatically the strings used by intrusion detection systems such as Snort. In other words, instead of having these strings be installed manually by security analysts, could a system automatically extract the suspicious strings? We ground the discussion in the context of detecting worm attack payloads.

The implementation techniques for security primitives described in this chapter (and the corresponding principles) are summarized in Figure 4.1.

| Number | Principle | Used In |
|---|---|---|
| P15 | Integrated string matching using Aho–Corasick | Snort |
| P3a, 5a | Approximate string match using min-wise hashing | Altavista |
| P3a | Path reconstruction using probabilistic marking | Edge sampling |
| P3a | Efficient packet logging via Bloom filters | SPIE |
| P3a | Worm detection by detecting frequent content | EarlyBird |

**FIGURE 4.1**

Principles used in the implementation of the various security primitives discussed in this chapter.

***Quick Reference Guide***

Sections 4.1.1 and 4.1.2 show how to speed up searching for *multiple* strings in packet payloads, a fundamental operation for a signature-based IDS. The Aho–Corasick algorithm of Section 4.1.1 can easily be implemented in hardware. While the traceback ideas in Section 4.4 are unlikely to be useful in the near future, the section introduces an important data structure, called a Bloom filter, for representing sets and also describes a hardware implementation. Bloom filters have found a variety of uses and should be part of the implementor's bag of tricks. Section 4.5 explains how signatures for attacks can be *automatically* computed, reducing the delay and difficulty required to have humans generate signatures.

## 4.1 SEARCHING FOR MULTIPLE STRINGS IN PACKET PAYLOADS

The first few sections tackle a problem of detecting an attack by searching for suspicious strings in payloads. A large number of attacks can be detected by their use of such strings. For example, packets that attempt to execute the Perl interpreter have *perl.exe* in their payload. For example, the arachNIDS database of vulnerabilities contains the following description.

An attempt was made to execute perl.exe. If the Perl interpreter is available to Web clients, it can be used to execute arbitrary commands on the Web server. This can be used to break into the server, obtain sensitive information, and potentially compromise the availability of the Web server and the machine it runs on. Many Web server administrators inadvertently place copies of the Perl interpreter

into their Web server script directories. If perl is executable from the cgi directory, then an attacker can execute arbitrary commands on the Web server.

This observation has led to a commonly used technique to detect attacks in so-called signature-based intrusion detection systems such as Snort. The idea is that a router or monitor has a set of rules, much like classifiers. However, the Snort rules go beyond classifiers by allowing a 5-tuple rule specifying the type of packet (e.g., port number equal to Web traffic) plus an arbitrary string that can appear anywhere in the packet payload.

Thus the Snort rule for the attempt to execute perl.exe will specify the protocol (TCP) and destination port (80 for Web) as well as the string "perl.exe" occurring anywhere in the payload. If a packet matches this rule, an alert is generated. Snort has 300 such augmented rules, with 300 possible strings to search for.

Early versions of Snort do string search by matching each packet against each Snort rule in turn. For each rule that matches in the classifier part, Snort runs a Boyer–Moore search on the corresponding string, potentially doing several string searches per packet. Since each scan through a packet is expensive, a natural question is: Can one search for all possible strings in one pass through packet?

There are two algorithms that can be used for this purpose: the Aho–Corasick algorithm and a modified algorithm due to Commentz-Walter, which we describe next.

### 4.1.1 Integrated String Matching Using Aho–Corasick

A trie can be used to search for a string that starts at a known position in a packet. Thus Figure 4.2 contains a trie built on the set of two strings "babar" and "barney"; both are well-known characters in children's literature. The trie is built on characters and not on arbitrary groups of bits. The characters in the text to be searched are used to follow pointers through the trie until a leaf string is found or until failure occurs.

The hard part, however, is looking for strings that can start anywhere in a packet payload. The naivest approach would be to assume the string starts at byte 1 of the payload and then traverses the trie. Then if a failure occurs, one could start again at the top of the trie with the character that starts at byte 2.

However, if packet bytes form several "near misses" with target strings, then for each possible starting position, the search can traverse close to the height of the trie. Thus if the payload has $L$ bytes and the trie has maximum height $h$, the algorithm can take $L \cdot h$ memory references.

For example, when searching for "babar" in the packet payload shown in Figure 4.2, the algorithm jogs merrily down the trie until it reaches the node corresponding to the second "a" in "babar." At that point the next packet byte is a "b" and not the "r" required to make progress in the trie. The naive approach would be to back up to the start of the trie and start the trie search again from the second byte "a" in the packet.

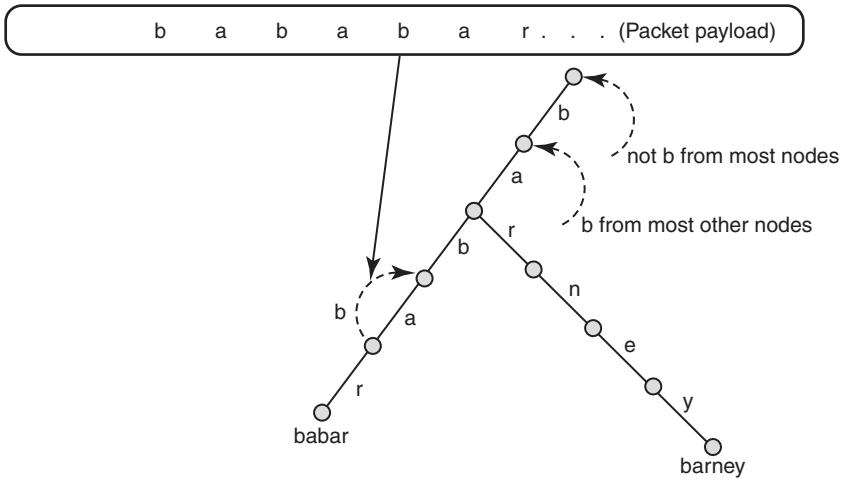


**FIGURE 4.2**

The Aho–Corasick algorithm builds an alphabetical trie on the set of strings to be searched for. A search for the string "barney" can be found by following the "b" pointer at the root, the "a" pointer at the next node, etc. More interestingly, the trie is augmented with failure pointers that prevent restarting at the top of the trie when failure occurs and a new attempt is made to match, shifted one position to the right.

However, it is not hard to see that backing up to the top is an obvious waste (**P1**) because the packet bytes examined so far in the search for "bababab" have "bab" as a suffix, which is a prefix of "babar." Thus, rather than back up to the top, one can precompute (much as in a grid of tries) a failure pointer corresponding to the failing "b" that allows the search to go directly to the node corresponding to path "bab" in the trie, as shown by the leftmost dotted arc in Figure 4.2.

Thus rather than have the fifth byte (a "b") lead to a null pointer, as it would in a normal trie, it contains a failure pointer that points back up the trie. Search now proceeds directly from this node using the sixth byte "a" (as opposed to the second byte) and leads after seven bytes to "babar."

Search is easy to do in hardware after the trie is precomputed. This is not hard to believe because the trie with failure pointers essentially forms a state machine. The Aho–Corasick algorithm has some complexity that ensues when one of the search strings, *R*, is a suffix of another search string, *S*. However, in the security context this can be avoided by relaxing the specification (**P3**). One can remove string *S* from the trie and later check whether the packet matched *R* or *S*.

Another concern is the potentially large number of pointers (256) in the Aho–Corasick trie. This can make it difficult to fit a trie for a large set of strings in cache (in software) or in SRAM (in hardware). One alternative is to use, say, Lulea-style encoding to compress the trie nodes.

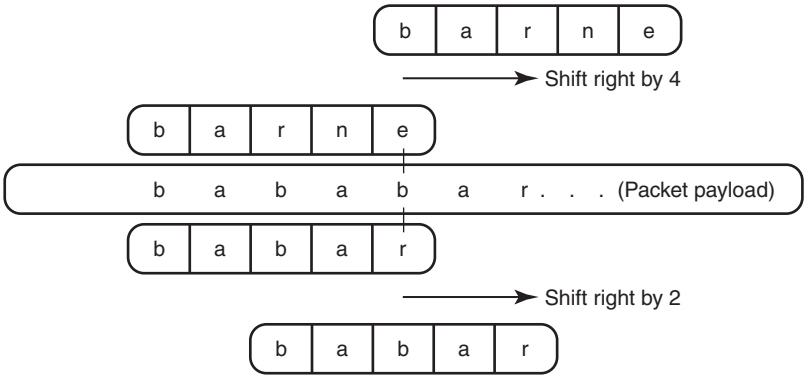


**FIGURE 4.3**

Integrated Boyer–Moore by shifting a character.

### 4.1.2 Integrated String Matching Using Boyer–Moore

The famous Boyer–Moore algorithm for *single*-string matching can be derived by realizing that there is an interesting degree of freedom that can be exploited (**P13**) in string matching: One can equally well start comparing the text and the target string from the last character as from the first.

Thus in Figure 4.3 the search starts with the fifth character of the packet, a "b," and matches it to the fifth character of, say, "babar" (shown below the packet), an "r." When this fails, one of the heuristics in the Boyer–Moore algorithm is to shift the search template of "babar" two characters to the right to match the rightmost occurrence of "b" in the template.[1] Boyer–Moore's claim to fame is that in practice it skips over a large number of characters, unlike, say, the Aho–Corasick algorithm.

To generalize Boyer–Moore to multiple strings, imagine that the algorithm concurrently compares the fifth character in the packet to the fifth character, "e," in the other string, "barney" (shown above the packet). If one were only doing Boyer–Moore with "barney," the "barney" search template would be shifted right by four characters to match the only "b" in barney.

When doing a search for both "barney" and "babar" concurrently, the obvious idea is to shift the search template by the smallest shift proposed by any string being compared for. Thus in this example, we shift the template by two characters and do a comparison next with the seventh character in the packet.

Doing a concurrent comparison with the last character in all the search strings may seem inefficient. This can be taken care of as follows. First, chop off all characters in all search strings beyond $L$, the shortest search string. Thus in Figure 4.3, $L$ is 5 and "barney" is chopped down to "barne" to align in length with "babar."

[1]There is a second heuristic in Boyer–Moore, but studies have shown that this simple Horspool variation works best in practice.

Having aligned all search string fragments to the same length, now build a trie starting *backwards* from the last character in the chopped strings. Thus, in the example of Figure 4.3 the root node of the trie would have an "e" pointer pointing toward "barne" and an "r" pointer pointing towards "babar." Thus comparing concurrently requires using only the current packet character to index into the trie node.

On success, the backwards trie keeps being traversed. On failure, the amount to be shifted is precomputed in the failure pointer. Finally, even if a backward search through the trie navigates successfully to a leaf, the fact that the ends may have been chopped off requires an epilogue, in terms of checking that the chopped-off characters also match. For reasonably small sets of strings, this method does better than Aho–Corasick.

The generalized Boyer–Moore was proposed by Commentz-Walter. The application to intrusion detection was proposed concurrently by Coit, Staniford, and McAlerney and Fisk and Varghese. The Fisk implementation has been ported to Snort.

Unfortunately, the performance improvement of using either Aho–Corasick or the integrated Boyer–Moore is minimal, because many real traces have only a few packets that match a large number of strings, enabling the naive method to do well. In fact, the new algorithms add somewhat more overhead due to slightly increased code complexity, which can exhibit cache effects.

While the code as it currently stands needs further improvement, it is clear that at least the Aho–Corasick version does produce a large improvement for *worst-case* traces, which may be crucial for a hardware implementation. The use of Aho–Corasick and integrated Boyer–Moore can be considered straightforward applications of efficient data structures (**P15**).

## 4.2 APPROXIMATE STRING MATCHING

This section briefly considers an even harder problem, that of approximately detecting strings in payloads. Thus instead of settling for an exact match or a prefix match, the specification now allows a few errors in the match. For example, with one insertion "perl.exe" should match "perl.exe" where the intruder may have added a character.

While the security implications of using the mechanisms described next need much more thought, the mechanisms themselves are powerful and should be part of the arsenal of designers of detection mechanisms.

The first simple idea can handle substitution errors. A *substitution error* is a replacement of one or more characters with others. For example, "parl.exe" can be obtained from "perl.exe" by substituting "a" for "e." One way to handle this is to search not for the complete string but for one or more random projections of the original string.

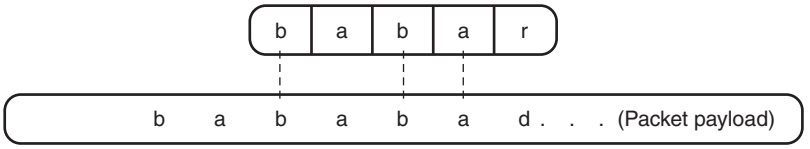


**FIGURE 4.4**

Checking for matching with a random projection of the target string "babar" allows the detecting of similar strings with substitution errors in the payload.

For example, in Figure 4.4, instead of searching for "babar" one could search for the first, third, and fourth characters in "babar." Thus the misspelled string "babad" will still be found. Of course, this particular projection will not find a misspelled string such as "rabad." To make it hard for an adversary, the scheme in general can use a small set of such random projections. This simple idea is generalized greatly in a set of papers on *locality-preserving hashing*.

Interestingly, the use of random projections may make it hard to efficiently shift one character to the right. One alternative is to replace the random projections by deterministic projections. For example, if one replaces every string by its two halves and places each half in an Aho–Corasick trie, then any one substitution error will be caught without slowing down the Aho–Corasick processing. However, the final efficiency will depend on the number of false alarms.

The simplest random projection idea, described earlier, does not work with insertions or deletions that can displace every character one or more steps to the left or right. One simple and powerful way of detecting whether two or more sets of characters, say, "abcef" and "abfecd," are similar is by computing their *resemblance*.

The resemblance of two sets of characters is the ratio of the size of their intersection to the size of their union. Intuitively, the higher the resemblance, the higher the similarity. By this definition, the resemblance of "abcef" and "abfecd" is 5/6 because they have five characters in common.

Unfortunately, resemblance per se does not take into account order, so "abcef" completely resembles "fecab." One way to fix this is to rewrite the sets with order numbers attached so that "abcef" becomes "1a2b3c4e5f" while "fecab" now becomes "1f2e3c4a5b." The resemblance, using pairs of characters as set elements instead of characters, is now nil. Another method that captures order in a more relaxed manner is to use shingles by forming the two sets to be compared using as elements all possible substrings of size $k$ of the two sets.

Resemblance is a nice idea, but it also needs a fast implementation. A naive implementation requires sorting both sets, which is expensive and takes large storage. Broder's idea is to quickly compare the two sets by computing a random (**P3a**, trade certainty for time) permutation on two sets. For example, the most practical permutation function on integers of size at most $m - 1$ is to compute $P(X) = ax + b \bmod m$, for random values of $a$ and $b$ and prime values of the modulus $m$.

For example, consider the two sets of integers {1, 3, 5} and {1, 7, 3}. Using the random permutation {$3x + 5 \bmod 11$}, the two sets become permuted to {8, 3, 9} and {8, 4, 3}. Notice that the minimum values of the two randomly permuted sets (i.e., 3) are the same.

Intuitively, it is easy to see that the higher the resemblance of the two sets, the higher the chance that a random permutation of the two sets will have the same minimum. Formally, this is because the two permuted sets will have the same minimum if and only if they contain the same element that gets mapped to the minimum in the permuted set. Since an ideal random permutation makes it equally likely for any element to be the minimum after permutation, the more elements the two sets have in common, the higher the probability that the two minimums match.

More precisely, the probability that two minimums match is equal to the resemblance. Thus one way to compute the resemblance of two sets is to use some number of random permutations (say, 16) and compute all 16 random permutations of the two sets. The fraction of these 16 permutations in which the two minimums match is a good estimate of the resemblance.

This idea was used by Broder to detect the similarity of Web documents. However, it is also quite feasible to implement at high link speeds. The chip must maintain, say, 16 registers to keep the current minimum using each of the 16 random hash functions. When a new character is read, the logic permutes the new character according to each of the 16 functions in parallel. Each of the 16 hash results is compared in parallel with the corresponding register, and the register value is replaced if the new value is smaller.

At the end, the 16 computed minima are compared in parallel against the 16 minima for the target set to compute a bitmap, where a bit is set for positions in which there is equality. Finally, the number of set bits is counted and divided by the size of the bitmap by shifting left by 4 bits. If the resemblance is over some specified threshold, some further processing is done.

Once again, the moral of this section is not that computing the resemblance is the solution to all problems (or in fact to any specific problem at this moment) but that fairly complex functions can be computed in hardware using multiple hash functions, randomization, and parallelism. Such solutions interplay principle **P5** (use parallel memories) and principle **P3a** (use randomization).

## 4.3 IP TRACEBACK VIA PROBABILISTIC MARKING

This section transitions from the problem of *detecting* an attack to *responding* to an attack. Response could involve a variety of tasks, from determining the source of the attack to stopping the attack by adding some checks at incoming routers.

The next two sections concentrate on *traceback*, an important aspect of response, given the ability of attackers to use forged IP source addresses. To understand the traceback problem it helps first to understand a canonical denial-of-service (DOS) attack that motivates the problem.

In one version of a DOS attack, called *SYN flooding*, wily Harry Hacker wakes up one morning looking for fun and games and decides to attack CNN. To do so he makes his computer fire off a large number of TCP connection requests to the CNN server, each with a different forged source address. The CNN server sends back a response to each request *R* and places *R* in a pending connection queue.

Assuming the source addresses do not exist or are not online, there is no response. This effect can be ensured by using random source addresses and by periodically resending connection requests. Eventually the server's pending-connection queue fills up. This denies service to innocent users like you who wish to read CNN news because the server can no longer accept connection requests.

Assume that each such denial-of-service attack has a traffic signature (e.g., too many TCP connection requests) that can be used to detect the onset of an attack. Given that it is difficult to shut off a public server, one way to respond to this attack is to trace such a denial-of service back to the originating source point despite the use of fake source addresses. This is the IP traceback problem.

The first and simplest systems approach (**P3**, relax system requirements) is to finesse the problem completely using help from routers. Observe that when Harry Hacker sitting in an IP subnetwork with prefix *S* sends a packet with fake source address *H*, the first router on the path can detect this fact if *H* does not match *S*. This would imply that Harry's packet cannot disguise its subnetworks, and offending packets can be traced at least to the right subnetwork.

There are two difficulties with this approach. First, it requires that edge routers do more processing with the source address. Second, it requires trusting edge routers to do this processing, which may be difficult to ensure if Harry Hacker has already compromised his ISP. There is little incentive for a local ISP to slow down performance with extra checks to prevent DOS attacks to a remote ISP.

A second and cruder systems approach is to have managers that detect an attack call their ISP, say, *A*. ISP *A* monitors traffic for a while and realizes these packets are coming from prior-hop ISP *B*, who is then called. *B* then traces the packets back to the prior-hop provider and so on until the path is traced. This is the solution used currently.

A better solution than *manual* tracing would be *automatic* tracing of the packet back to the source. Assume one can modify routers for now. Then packet tracing can be trivially achieved by having each router in the path of a packet *P* write its router IP address in sequence into *P*'s header. However, given common route lengths of 10, this would be a large overhead (40 bytes for 10 router IDs), especially for minimum-size acknowledgments. Besides the overhead, there is the problem of modifying IP headers to add fields for path tracing. It may be easier to steal a small number of unused message bits.

This leads to the following problem. Assuming router modifications are possible, find a way to trace the path of an attack by marking as few bits as possible in a packet's header.

For a single-packet attack, this is very difficult in an information theoretic sense. Clearly, it is impossible to construct a path of 10 32-bit router IDs from, say, a 2-byte mark in a packet. One can't make a silk purse from a sow's ear.

However, in the systems context one can optimize the expected case (**P11**), since most interesting attacks consist of hundreds of packets at least. Assuming they are all coming from the same physical source, the victim can shift the path computation over time (**P2**) by making each mark contribute a piece of the path information.

Let's start by assuming a single 32-bit field in a packet that can hold a single router ID. How are the routers on the path to synchronize access to the field so that each router ID gets a chance, over a stream of packets, to place its ID in the field?

A naive solution is shown in Figure 4.5. The basic idea is that each router independently writes its ID into a *single* node ID field in the packet with probability $p$, possibly overwriting a previous router's ID. Thus in Figure 4.5, the packet already has $R1$ in it and can be overwritten by $R3$ to $R1$ with probability $p$.

The hope, however, is that over a large sequence of packets from the attacker to the victim, every router ID in the path will get a chance to place its ID without being overwritten. Finally, the victim can sort the received IDs by the number of samples. Intuitively, the nodes closer to the victim should have more samples, but one has to allow for random variation.

The two problems with this naive approach are that too many samples (i.e., attack packets) are needed to deal with random variation in inferring order, and the attacker, knowing this scheme, can place malicious marks in the packet to fool the reconstruction scheme into believing that fictitious nodes are close to the victim because they receive extra marks.

To foil this threat, $p$ must be large, say, 0.51. But in this case, the number of packets required to receive the router IDs far away from the victim becomes very large. For example, with $p = 0.5$ and a path of length $L = 15$, the number of packets required is the reciprocal of the probability that the router farthest from the victim sends a mark that survives. This is $p(1-p)^{L-1} = 2^{-15}$, because it requires the farthest router to put a mark and the remaining $L - 1$ routers not to. Thus the average number of packets for this to happen is $1/2^{-15} = 32{,}000$. Attacks have a number of packets, but not necessarily this many.

The straightforward lesson from the naive solution is that randomization is good for synchronization (to allow routers to independently synchronize access to the single node ID field) but not to reconstruct order. The simplest solution to this

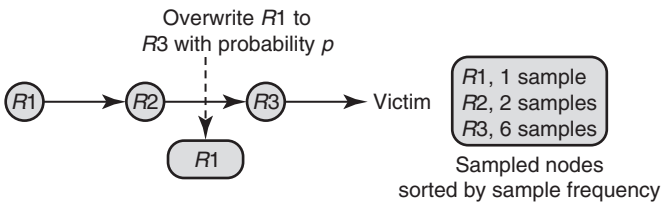


**FIGURE 4.5**

Reconstructing an attack path by having each router stamp its ID independently, with probability $p$, into a single node ID field. The receiver reconstructs order by sorting, assuming that closer routers will produce more samples.

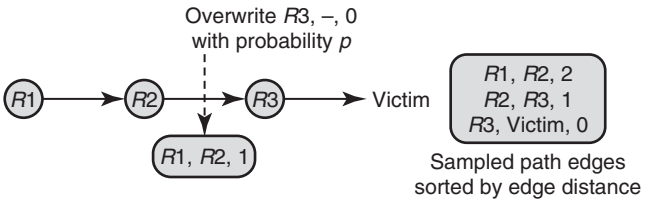


**FIGURE 4.6**

Edge sampling improves on node sampling by sampling edges and not nodes. This allows trivial order reconstruction based on edge distance and not sample frequency.

problem is to use a hop count (the attacker can initialize each packet with a different TTL, making the TTL hard to use) as well as a node ID. But a hop count by itself can be confusing if there are multiple attacks going on. Clearly a mark of node $X$ with hop count 2 may correspond to a different attack path from a mark of node $Y$ with hop count 1.

The solution provided in the seminal paper avoids the aliasing due to hop counts by conceptually starting with a pair of consecutive node IDs and a hop count to form a triple $(R, S, h)$, as shown in Figure 4.6.

When a router $R$ receives a packet with triple $(X, Y, h)$, $R$ generates a random number between 0 and 1. If the number is less than the sampling probability $p$, router $R$ writes its own ID into the mark triple, rewriting it as $(R, -, 0)$, where the $-$ character indicates that the next router in the path has still to be determined. If the random number is greater than $p$, then $R$ must maintain the integrity of the previously written mark. If $h = 0$, $R$ writes $R$ to the second field because $R$ is the next router after the writer of the mark. Finally, if the random number is greater than $p$, $R$ increments $h$.

It should be clear that by assuming that every edge gets sampled once, the victim can reconstruct the path. Note also that the attacker can only add fictitious nodes to the start of the path. But how many packets are required to find all edges? Given that ordering is explicit, one can use arbitrary values of $p$.

In particular, if $p$ is approximately $1/L$, where $L$ is the path length to the farthest router, the probability we computed before of the farthest router sending an edge mark that survives becomes $p(1 - p)^{L-1} \approx p/(1 - p)e$, where $e$ is the base of natural logarithms. For example, for $p = 1/25$, this is roughly 1/70, which is fairly large compared to the earlier attempt.

What is even nicer is that if we choose $p = 1/50$ based on the largest path lengths encountered in practice on the Internet (say, 50), the probability does not grow much smaller even for much smaller path lengths. This makes it easy to reconstruct the path with hundreds of packets as opposed to thousands.

Finally, one can get rid of obvious waste (**P1**) and avoid the need for two node IDs by storing only the Exclusive-OR of the two fields in a single field. Working backwards from the last router ID known to the victim, one can Exclusive-OR with the previous edge mark to get the next router in the path, and so on. Finally,

by viewing each node as consisting of a sequence of a number of "pseudonodes," each with a small fragment (say, 8 bits) of the node's ID, one can reduce the mark length to around 16 bits total.

## 4.4 IP TRACEBACK VIA LOGGING

A problem with the edge-sampling approach of the previous section is that it requires changes to the IP header to update marks and does not work for single-packet attacks like the Teardrop attack. The following approach, traceback via logging, avoids both problems by adding more storage at routers to maintain a compressed packet log.

As motivations, neither of the difficulties the logging approach gets around are very compelling. This is because the logging approach still requires modifying router forwarding, even though it requires no header modification. This is due to the difficulty of convincing vendors (who have already committed forwarding paths to silicon) and ISPs (who wish to preserve equipment for, say, 5 years) to make changes. Similarly, single-packet attacks are not very common and can often be filtered directly by routers.

However, the idea of maintaining compressed searchable packet logs may be useful as a general building block. It could be used, more generally, for, say, a network monitor that wishes to maintain such logs for forensics after attacks. But even more importantly it introduces an important technique called *Bloom filters*.

Given an efficient packet log at each router, the high-level idea for traceback is shown in Figure 4.7. The victim $V$ first detects an attack packet $P$; it then queries all its neighboring routers, say, $R_8$ and $R_9$, to see whether any of them have $P$ in their log of recently sent packets. When $R_9$ replies in the affirmative, the search moves on to $R_9$, who asks its sole neighbor, $R_7$. Then $R_7$ asks its neighbors $R_5$ and $R_4$, and the search moves backward to $A$.

The simplest way to implement a log is to reuse one of the techniques in trajectory sampling. Instead of logging a packet we log a 32-bit hash of invariant content (i.e., exclude fields that change from hop to hop, such as the TTL) of the packet. However, 32 bits per packet for all the packets sent in the last 10 minutes is still huge at 10 Gbps. Bloom filters, described next, allow a large reduction to around 5 bits per packet.

### 4.4.1 Bloom Filters

Start by observing that querying either a packet log or a table of allowed users is a *set membership query*, which is easily implemented by a hash table. For example, in a different security context, if John and Cathy are allowed users and we wish to check if Jonas is an allowed user, we can use a hash table that stores John and Cathy's IDs but not Jona's.

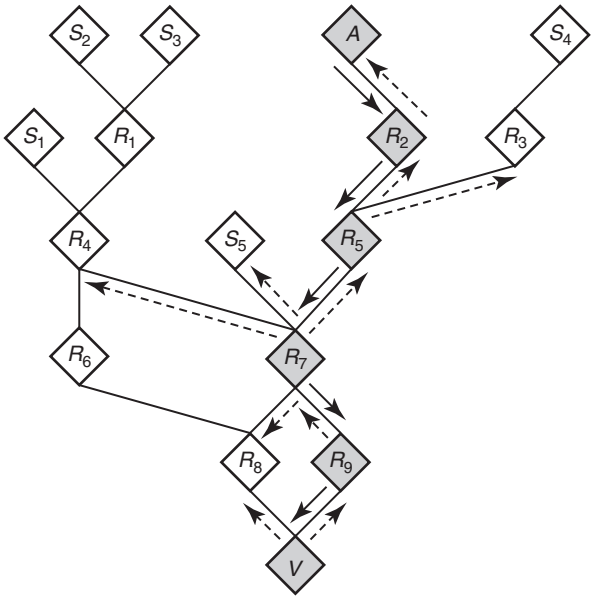


**FIGURE 4.7**

Using a packet log to trace an attack packet *P* backwards from the victim *V* to the attacker *A* by having the currently traced node ask all its neighbors (the dotted lines) if they have seen *P* (solid line).

Checking for Jonas requires hashing Jonas's ID into the hash table and following any lists at that entry. To handle collisions, each hash table entry must contain a list of IDs of all users that hash into that bucket. This requires at least *W* bits per allowed user, where *W* is the length of each user ID. In general, to implement a hash table for a set of identifiers requires at least *W* bits per identifier, where *W* is the length of the smallest identifier.

Bloom filters, shown in Figure 4.8, allow one to reduce the amount of memory for set membership to a few bits per set element. The idea is to keep a bitmap of size, say, 5*N*, where *N* is the number of set elements. Before elements are inserted, all bits in the bitmap are cleared.

For each element in the set, its ID is hashed using *k* independent hash functions (two in Figure 4.8, *H*1 and *H*2) to determine bit positions in the bitmap to set. Thus in the case of a set of valid users in Figure 4.8, ID John hashes into the second and next-to-last bit positions. ID Cathy hashes into one position in the middle and also into one of John's positions. If two IDs hash to the same position, the bit remains set.

Finally, when searching to see if a specified element (say, Jonas) is in the set, Jonas is hashed using all the *k* hash functions. Jonas is assumed to be in the set if

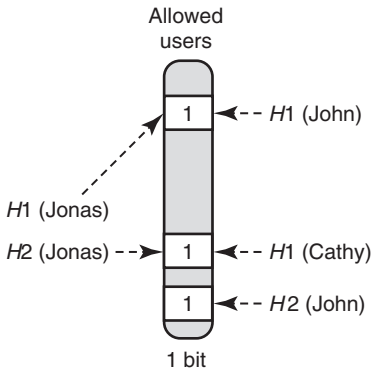


**FIGURE 4.8**

A Bloom filter represents a set element by setting *k* bits in a bitmap using *k* independent hash functions applied to the element. Thus the element John sets the second (using *H*1) and next-to-last (using *H*2) bits. When searching for Jonas, Jonas is considered a member of the set only if all bit positions hashed to by Jonas have set bits.

all the bits hashed into by Jonas are set. Of course, there is some chance that Jonas may hash into the position already set by, say, Cathy and one by John (see Figure 4.8). Thus there is a chance of what is called a *false positive*: answering the membership query positively when the member is not in the set.

Notice that the trick that makes Bloom filters possible is relaxing the specification (**P3**). A normal hash table, which requires $W$ bits per ID, does not make errors! Reducing to 5 bits per ID requires allowing errors; however, the percentage of errors is small. In particular, if there is an attack tree and set elements are hashed packet values, as in Figure 4.7, false positives mean only occasionally barking up the wrong tree branch(es).

More precisely, the false-positive rate for an $m$-size bitmap to store $n$ members using $k$ hash functions is

$$(1 - (1 - 1/m)^{kn})^k \approx (1 - e^{kn/m})^k$$

The equation is not as complicated as it may appear: $(1 - 1/m)^{kn}$ is the probability that any bit is *not* set, given $n$ elements that each hashes $k$ times to any of $m$ bit positions. Finally, to get a false positive, all of the $k$ bit positions hashed onto by the ID that causes a false positive must be set.

Using this equation, it is easy to see that for $k = 3$ (three independent hash functions) and 5 bits per member ($m/n = 5$), the false-positive rate is roughly 1%. The false-positive rate can be improved up to a point by using more hash functions and by increasing the bitmap size.

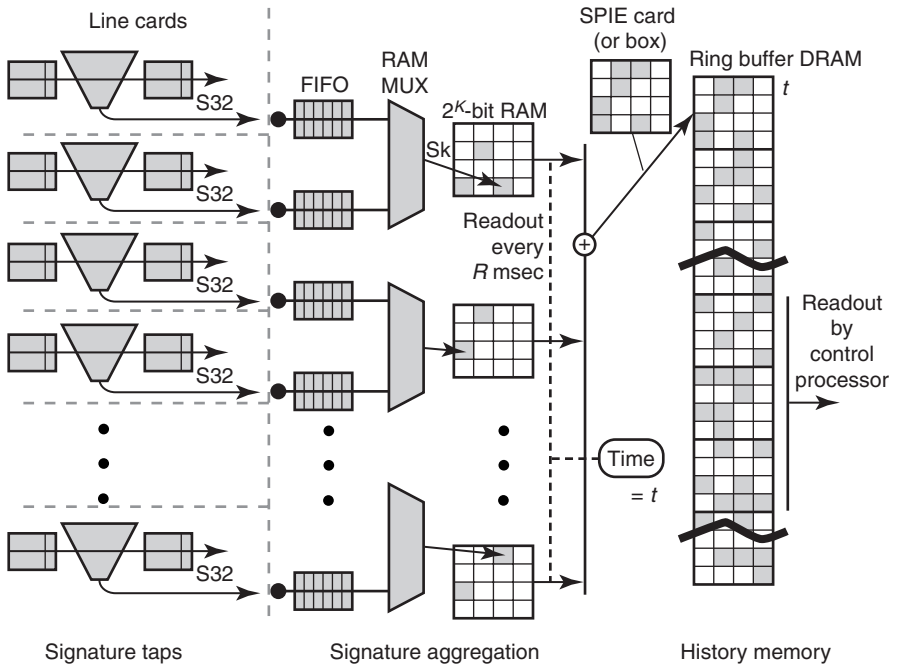


**FIGURE 4.9**

Hardware implementation of packet logging using Bloom filters. Note the use of two-level memory: SRAM for random read-modify-writes and DRAM for large row writes.

### 4.4.2 Bloom Filter Implementation of Packet Logging

The Bloom filter implementation of packet logging in the SPIE system is shown in Figure 4.9 (the picture is courtesy of Sanchez et al.). Each line card calculates a 32-bit hash digest of the packet and places it in a FIFO queue. To save costs, several line cards share, via a RAM multiplexor, a fast SRAM containing the Bloom filter bitmap.

As in the case of counters, one can combine the best features of SRAM and DRAM to reduce expense. One needs to use SRAM for fast front-end *random access* to the bitmap. Unfortunately, the expense of SRAM would allow storing only a small number of packets. To allow a larger amount, the Bloom filter bitmaps in SRAM are periodically read out to a large DRAM ring buffer. Because these are no longer random writes to bits, the write to DRAM can be written in DRAM pages or rows, which provide sufficient memory bandwidth.

## 4.5 DETECTING WORMS

It would be remiss to end this chapter without paying some attention to the problem of detecting worms. A worm (such as Code Red, Nimda, Slammer) begins

with an exploit sent by an attacker to take over a machine. The exploit is typically a buffer overflow attack, which is caused by sending a packet (or packets) containing a field that has more data than can be handled by the buffer allocated by the receiver for the field. If the receiver implementation is careless, the extra data beyond the allocated buffer size can overwrite key machine parameters, such as the return address on the stack.

Thus with some effort, a buffer overflow can allow the attacking machine to run code on the attacked machine. The new code then picks several random IP addresses[2] and sends similar packets to these new victims. Even if only a small fraction of IP addresses responds to these attacks, the worm spreads rapidly.

Current worm detection technology is both *retroactive* (i.e., only after a new worm is first detected and analyzed by a human, a process that can take days, can the containment process be initiated) and *manual* (i.e., requires human intervention to identify the signature of a new worm). Such technology is exemplified by Code Red and Slammer, which took days of human effort to identify, following which containment strategies were applied in the form of turning off ports, applying patches, and doing signature-based filtering in routers and intrusion detection systems.

There are difficulties with these current technologies.

1. *Slow Response:* There is a proverb that talks about locking the stable door after the horse has escaped. Current technologies fit this paradigm because by the time the worm containment strategies are initiated, the worm has already infected much of the network.
2. *Constant Effort:* Every new worm requires a major amount of human work to identify, post advisories, and finally take action to contain the worm. Unfortunately, all evidence seems to indicate that there is no shortage of new exploits. And worse, simple binary rewriting and other modifications of existing attacks can get around simple signature-based blocking (as in Snort).

Thus there is a pressing need for a new worm detection and containment strategy that is real time (and hence can contain the worm before it can infect a significant fraction of the network) and is able to deal with new worms with a minimum of human intervention (some human intervention is probably unavoidable to at least catalog detected worms, do forensics, and fine-tune automatic mechanisms). In particular, the detection system should be *content agnostic*. The detection system should not rely on external, manually supplied input of worm signatures.

Instead, the system should *automatically* extract worm signatures, even for new worms that may arise in the future.

[2] By contrast, a *virus* requires user intervention, such as opening an attachment, to take over the user machine. Viruses also typically spread by using known addresses, such as those in the mail address book, rather than random probing.

Can network algorithmics speak to this problem? We believe it can. First, we observe that the only way to detect new worms and old worms with the same mechanism is to abstract the basic properties of worms.

As a first approximation, define a worm to have the following abstract features, which are indeed discernible in all the worms we know, even ones with such varying features as Code Red (massive payload, uses TCP, and attacks on the well-known HTTP port) and MS SQL Slammer (minimal payload, uses UDP, and attacks on the lesser-known MS SQL port).

1. *Large Volume of Identical Traffic:* These worms have the property that at least at an intermediate stage (after an initial priming period but before full infection), the volume of traffic (aggregated across all sources and destinations) carrying the worm is a significant fraction of the network bandwidth.
2. *Rising Infection Levels:* The number of infected sources participating in the attack steadily increases.
3. *Random Probing:* An infected source spreads infection by attempting to communicate to random IP addresses at a fixed port to probe for vulnerable services.

Note that detecting all three of these features may be crucial to avoid false positives. For example, a popular mailing list or a flash crowd could have the first feature but not the third.

An algorithmics approach for worm detection would naturally lead to the following detection strategy, which automatically detects each of these abstract features with low memory and small amounts of processing, works with asymmetric flows, and does not use active probing. The high-level mechanisms[3] are:

1. *Identify Large Flows in Real Time with Small Amounts of Memory*: Mechanisms can be described to identify flows with large traffic volumes for any definition of a flow (e.g., sources, destinations). A simple twist on this definition is to realize that the content of a packet (or, more efficiently, a hash of the content) can be a valid flow identifier, which by prior work can identify in real time (and with low memory) a high volume of repeated content. An even more specific idea (which distinguishes worms from valid traffic such as peer-to-peer) is to compute a hash based on the content as well as the destination port (which remains invariant for a worm).
2. *Count the Number of Sources:* Mechanisms can be described using simple bitmaps of small size to estimate the number of sources on a link using small amounts of memory and processing. These mechanisms can easily be used to count sources corresponding to high traffic volumes identified by the previous mechanism.

[3]Each of these mechanisms needs to be modulated to handle some special cases, but we prefer to present the main idea untarnished with extraneous details.

3. *Determine Random Probing by Counting the Number of Connection Attempts to Unused Portions of the IP Address:* One could keep a simple compact representation of portions of the IP address space known to be unused. One example is the so-called Bogon list, which lists unused 8-bit prefixes (can be stored as a bitmap of size 256). A second example is a secret space of IP addresses (can be stored as a single prefix) known to an ISP to be unused. A third is a set of unused 32-bit addresses (can be stored as a Bloom filter).

Of course, worm authors could defeat this detection scheme by violating any of these assumptions. For example, a worm author could defeat Assumption 1 by using a very slow infection rate and by mutating content frequently. Assumption 3 could be defeated using addresses known to be used. For each such attack there are possible countermeasures. More importantly, the scheme described seems certain to detect at least all existing worms we know of, though they differ greatly in their semantics. In initial experiments at UCSD as part of what we call the EarlyBird system, we also found very few false positives where the detection mechanisms complained about innocuous traffic.

## 4.6 CONCLUSION

Returning to Marcus Ranum's quote at the start of this chapter, hacking must be exciting for hackers and scary for network administrators, who are clearly on different sides of the battlements. However, hacking is also an exciting phenomenon for practitioners of network algorithmics—there is just so much to do. Compared to more limited areas, such as accounting and packet lookups, where the basic tasks have been frozen for several years, the creativity and persistence of hackers promise to produce interesting problems for years to come.

In terms of technology currently used, the set string-matching algorithms seem useful and may be ignored by current products. However, other varieties of string matching, such as regular expression matches, are in use. While the approximate matching techniques are somewhat speculative in terms of current applications, past history indicates they may be useful in the future.

Second, the traceback solutions only represent imaginative approaches to the problem. Their requirements for drastic changes to router forwarding make them unlikely to be used for current deployment as compared to techniques that work in the control plane. Despite this pessimistic assessment, the underlying techniques seem much more generally useful.

For example, sampling with a probability inversely proportional to a rough upper bound on the distance is useful for efficiently collecting input from each of a number of participants without explicit coordination. Similarly, Bloom filters are useful to reduce the size of hash tables to 5 bits per entry, at the cost of a small probability of false positives. Given their beauty and potential for high-speed

implementation, such techniques should undoubtedly be part of the designer's bag of tricks.

Finally, we described our approach to content-agnostic worm detection using algorithmic techniques. The solution combines existing mechanisms described earlier in this book. While the experimental results on our new method are still preliminary, we hope this example gives the reader some glimpse into the possible applications of algorithmics to the scary and exciting field of network security. Figure 4.1 presents a summary of the techniques used in this chapter, together with the major principles involved.