600.424 Final Project:
Comparison of Authenticated Traceback Schemes

Lucas Ballard
lucas@jhu.edu

Peter Kimball
kimball@jhu.edu

December 8, 2003
# Contents

1 Goals ................................ ................................ .................................... 3
2 Algorithms and Implementation ................................ ...................................... 3
   2.1 Simnet ................................ ................................ ...................................... 3
   2.2 Ballard/Kimball ................................ ................................ ............................. 3
      2.2.1 An Attack ................................ ................................ ................................ 4
   2.3 Song ................................ ................................ .......................................... 5
   2.4 Adler ................................ ................................ .......................................... 6
      2.4.1 Issues with the Algorithm ................................ ................................ ...... 6
      2.4.2 Implementation ................................ ................................ ....................... 7
3 Topologies ................................ ................................ ...................................... 8
4 Results ................................ ................................ ......................................... 9
   4.1 Path Reconstruction Metrics ................................ ................................ .... 9
   4.1.1 Victim Reconstruction Time ................................ ................................ .. 9
   4.1.2 Packets Required for Accurate Traces ................................ .................... 11
   4.2 Router Overhead ................................ ................................ ............................ 13
   4.3 Impact on Regular Traffic ................................ ................................ .......... 14
   4.4 Robustness against Attacker-Marked Traffic ................................ ............ 16
      4.4.1 Adler ................................ ................................ ....................................... 16
      4.4.2 Song ................................ ................................ ....................................... 18
   4.5 Out-of-band communication ................................ ................................ ....... 18
5 Conclusion ................................ ................................ ..................................... 19
1 Goals

DoS and DDoS attacks are prevalent on the Internet. Such attacks are relatively easy to implement, can be very damaging and allow a method of attack in which the initiator can escape without being tracked. It is therefore quite interesting to discover a way to trace such attacks to their sources. Many current schemes implement forms of probabilistic packet marking (PPM). [1, 3, 4, 6, 7, 8] This project compares three different traceback algorithms: That presented by Adler in [1], the Advanced Marking scheme proposed by Song et. al. in [8] and one proposed by us. Each of these algorithms are resistant to malicious adjustments by the attacker to the bits used in the algorithm. The goal of this project is to compare these algorithms in terms of the number of bits used per packet, reconstruction time, router and end-host overhead, impact on regular traffic, and out-of-band communication.

2 Algorithms and Implementation

This section will describe the basics of each algorithm and the implementation in Simnet. We will pay particular attention to our algorithm and Adler’s algorithm as the reader is probably not as familiar with ours as the other two and Adler’s scheme is significantly more complicated than the other two.

2.1 Simnet

We created a class called Project_IP_Packet that extends Packet. We added two fields to this packet to make it more “realistic,” the tos and flags fields. We also enforced mechanisms that guaranteed we only used as many bits as specified by the IP protocol. Thus, although all of our fields are of type int, we only use 5 bits of the tos field and 16 of the ipid field. Originally we only modified our code and PacketGenerator to handle Project_IP_Packets, however we realized that this made it so other types of traffic (e.g., RSTs) were not analyzed by our algorithms. Thus, we had to replace all instances of IP_Packet with Project_IP_Packet. If we had to do this again, we may have done things differently in this regard.

2.2 Ballard/Kimball

Our algorithm is a hybrid algorithm, using the S/Key key generation algorithm described in Song [8] in addition to several novel elements. Dean [3] shows us that there are twenty-five bits readily available in the IP header, to which we add the 32-bit source address field. The TOS field, IPID field, and one bit of the flags field were available for use, due to their infrequent use in regular IP traffic. Though we realize that it will incur damage to regular traffic, we felt that the appropriation of the source address field is an acceptable means of gaining the extra bits we desired for our algorithm because this address is often changed by attackers anyway. We use these fifty-seven bits to store a 32-bit router address field, a 20-bit hash field, and a 5-bit distance field. Song [8] claims that the 5-bit distance field will correctly represent most internet distances and we therefore chose that value for our distance field. Since we wanted a non-hashed representation of the router’s address, we used the 32-bit source address as a 32-bit router address, allowing us to forgo the address-guessing steps of the Song reconstruction algorithm. This left us with twenty bits, which we determined would be sufficiently large enough to prevent hash collisions between routers during a flood. Since the hash only contains the IP address of the marking router, it is possible to see marked packets from over 1,000,000 routers before we must deal with hash collisions. This is a significant improvement over the hash collisions which happen every 2,048 packets due to the 11-bit hash field used by Song. Since our algorithm uses unique hashes for marking routers, our collision rate is dependent on the number of routers from which we receive packets, while the collision rate in Song’s packet-unique hashing scheme is dependent on the number of observed packets.
When we mark a packet, we insert the router’s address into the source address of the packet, place a keyed hash of the router’s address into the hash field, and set the distance field to be zero. If a router is not marking a packet and sees a marked packet with distance zero, it performs an XOR between its IP and the IP stored in the packet’s source address and increments the distance field. If a non-marking router sees a marked packet with a distance greater than zero, it simply increments the distance.

In order to reconstruct the attack path, we mark the receipt time of marked packets and starting at depth one, we retrieve the current key for the router identified in the source field. We can then calculate the previous keys as needed using the S/Key algorithm. We construct a time-appropriate hash for the router, using the generated keys and compare it to the hash stored in the received packet. If we have a match, the router is verified as being on the path. We repeat this for greater depths, though for depths greater than one we must XOR the source field value with the value from the previous-distance router in order to retrieve a valid router IP.

The implementation of our algorithm takes place primarily through two classes, BallardKimballRouter and BallardKimballTraceback. At initialization our router generates a seed key and then uses that key to build a key chain. For the sake of simplicity, we did not sign and publish this seed key, as is specified in the S/Key description. After creating the key chain, we start two repeating TimerTasks, one of which increments our index in the key chain and the other of which publishes old keys. The key-incrementing task is called once per time-slice and simply increases our index in the key chain and recalculates the hash which will be applied to packets passing through the router during the time-slice. The key-publishing task simply writes a key to a file after it has been out of use for two time-slices. We use files rather than direct communication with the victim for simplicity’s sake. Our router implementation overrides the forward() method of simnet.Router, which is passed a Project.IP.Packet instead of a simnet.IP.Packet. The forward method uses a coin-flip to decide whether or not to mark the packet, with the probability controlled by a static marking rate. Marking takes place as described above, though the actual marking is accomplished by methods in Project.IP.Packet.

Our reconstruction algorithm is significantly more complicated than our marking algorithm, though significantly less complicated than the reconstruction algorithm required for our implementation of the Song algorithm. When we are in traceback mode and receive a marked packet, we remove the pertinent data and place that data along with a timestamp in a data structure indexed to the packet’s depth. We then overwrite the source address with our own address, so that replies are not sent to erroneous addresses (in affect, we drop the packet). At reconstruction time we cycle through the data structure containing the information from the packets we received, starting with distance zero. We then try and authenticate a packet from this distance by performing an XOR of the source address with the address of the previous router, if appropriate. We then retrieve the time-appropriate key for the router which initially marked this packet by reading in the last entry in the key chain file for the marking router. This key is then used to create a new hash of the marking router’s address and that hash is compared to the stored hash. If they match, then we add the edge to our graph, or increase the edge-count if it is already in the graph. After we have cycled through all the depths for which we have received packets, we print out our attack path graph.

2.2.1 An Attack

Since our algorithm stores only router-specific information in the keyed hash, this hash can be applied to more than one packet. This is a strength of our algorithm because it reduces the number of hash calculations the router must perform, yet it is also a weakness because once this hash is known, an attacker may insert packets with valid hashes into a stream which should not include those hashes. An attacker can extend the attack graph beyond their location, provided they can retrieve from and insert information into the stream during a single time-slice.
The actual attack involves an attacker and an accomplice. The accomplice should be on the opposite side of the router from the attacker, with the router containing the hash the attacker wishes to insert into their attack stream. The accomplice should flood the attacker so that there is a high likelihood of the attacker receiving a packet marked by the router. The attacker can then insert packets with this marking into the attack stream, with an insertion likelihood that matches the marking rate defined for the scheme. Since these markings will validate at the victim only during the time-slice defined by the router, false edges will validate only if the time-slice is not small enough to prevent traffic insertion.

We feel that the small 1-second time-slice defined in our algorithm along with the extremely low marking rate will prevent the majority of these attacks as attackers will find it hard to not only retrieve the hash from the flood, but also to insert the marking into their traffic at the appropriate rate, all within the small time-slice. Even if an attacker were to accomplish this task, they would still lie on the attack graph, exposing themselves to discovery.

2.3 Song

We feel that the Song algorithm is best described in their paper [8] and refer the reader to that paper for any details not illuminated within this explanation of our implementation of the algorithm.

Our implementation of the Song algorithm is accomplished primarily in SongRouter.java and SongTraceback.java. The router implementation is very similar to the implementation described above for our algorithm. The main exception here is that the key-incrementing task for SongRouter does not automatically calculate the hash which the router will use, because that hash is packet-dependent and must be re-calculated each time a packet is marked. The other significant difference is that the marking done by the second router involves calculating the router’s hash for the packet and then performing an XOR of that hash with the existing hash and writing the new value to the appropriate field. The address of the router does not appear anywhere in the packet in a non-hashed form. As with our algorithm, the actual marking is accomplished in Project_IP_Packet for reasons of simplicity. It is worth noting at this point that the eleven bit hash used by Song represents the address of the source, destination, and marking router of a packet, while the twenty bit hash in our algorithm represents only the address of the marking router.

Attack path reconstruction for the Song algorithm is an extremely time and code intensive undertaking. Like the traceback implementation for our algorithm, we store packet information and an arrival-timestamp in a distance-based data structure upon packet receipt. At traceback time, we establish the current network topology by retrieving the identities of all the routers from simnet.Simnet and performing a traceroute to each in turn. Their distances are marked in a data structure for future reference. Like our traceback implementation, at traceback time we parse through this structure and attempt to verify edges starting with the ones closest to the victim. The edges at distance 0 have not been subject to an XOR operation with another edge, so that we immediately retrieve a router’s hash from the packet. We then retrieve the keys for all routers at the marked distance and store them as IPAddress which links the router address to the current key for that router. After we have an array of keys we cycle through them and use each key to build a hash of the source and destination specified in the packet, as well as the router address specified in the key we are currently examining. If this hash matches the hash stored in the packet, we can verify that edge and add it to the graph. If the arrival time linked to the packet does not fall within the time-slice for which the router’s key was valid, we hash the key according to the S/Key specification until the correct time-slice is reached. Once we get past distance 0 and are performing XOR operations with the previous distance, we must remember to retrieve the correct-time key for the previous distance and re-build the hash for that distance before removing it through the XOR operation. This makes sure that we are not attempting to use a hash which was built during a previous time-slice, otherwise we would have to receive packets from all routers during a single time-slice in order for the consecutive hashes to be applied correctly. After we have cycled through all the depths for which we have received packets, we print
out our attack path graph.

2.4 Adler

The most difficult aspect of implementing the scheme proposed by Adler [1] was understanding the paper; the algorithm itself is relatively straightforward to code. There were two aspects of the paper that posed the most difficulty: an error in one of the functions he described and the lack of discussion associated with a possible real-world implementation. We will briefly discuss those issues and our solutions, and then our implementation of his algorithm in Simnet.

2.4.1 Issues with the Algorithm

The first problem arose in the implementation of his “bucket” protocol. (What we refer to as his “bucket” protocol is what he calls the Multi-Bit protocol. We chose the former name because our implementation of his Single-Bit protocol required multiple bits due to the fact that we extended beyond his binary tree model.) The problem was in the definition of his function \( z(n, k) = 1 + \left\lfloor \frac{n - k}{d} \right\rfloor \) which is implemented as \( z(k) \) in AdlerTraceback.java. This function is supposed to determine the number of integers between 1 and \( n \) inclusive, that are congruent to \( d \) mod \( k \). While his function is correct when \( d \) does not divide \( n - k \) (as is generally the case), it is incorrect otherwise (If \( n = 16, k = 0 \) and \( d = 4 \) then the function returns 5, however, there are only 4 integers in \([1, 16]\) that are congruent to 0 mod 4). To fix this we simply added a test to our implementation of \( z(k) \). This error yielded many problems, as the size of the sub-arrays in the “bucket” reconstruction algorithm are dependent upon this number. Not only was our code attempting to access memory that was not allocated to us, the mapping from the index in the sub-array to index in the overall array was incorrect (\( A' \) and \( A \) respectively on page 9 of [1]), so our binary expansion of the decimal probabilities was inaccurate.

The second problem posed a greater design challenge than the first. We had to come up with our own method of handling routers that had more than two interfaces. Adler briefly mentioned that we could encode the interface number as a Shannon code with a uniform distribution over the interfaces of the router [1]. This did not make sense to us because the lower bound for the length of a Shannon code is given by the entropy of the source [2]. If the distribution is uniform for the random variable \( X \) then the entropy is

\[
H(X) = -\sum_{x \in X} p(x) \log_2 p(x) = -\sum_{x \in X} \frac{1}{|X|} \log_2 \frac{1}{|X|} = -\log_2 \frac{1}{|X|} = \log_2 |X|
\]

Thus, if we considered each of \( L \) interfaces with the same probability, the Shannon code could do no better than \( \log_2 L \) bits, which is no better (and more complicated) than simply coding the interface number in binary. We considered trying use a non-uniform probability distribution over the interfaces, for instance, based on the number of packets an interface received. However, this could allow attackers to change the encoding of an interface by sending more packets to a different interface during an attack. If the encoding of an interface number changed during the attack, then the reconstruction algorithm could fail. Thus, we decided to encode each interface in binary and assign an upper-bound (16) on the number of interfaces in our topology. Obviously, this poses an issue for real-world implementation.

The problem still remained of how to allow the victim to determine the interface number that a router was sending. Adler made no mention of this, perhaps he thought it too obvious. We elected to extend the basic algorithm by keeping a counter for each of the \( \log_2 L \) bits marked by the routers (instead of just one) on the victim and recover part of the interface number for each router based on these counters. By concatenating the \( i^{th} \) bit in the binary expansion of each counter, we are able to determine the interface number of the router at distance \( i \). We were not able to come up with an encoding that would allow us to trace multiple paths of attack.
2.4.2 Implementation

The code that implements Adler’s scheme resides in two files: AdlerTraceback.java and AdlerRouter.java. These files implement the “bucket” algorithm, as described on page 9 of the paper. The simpler version of the algorithm that requires more packets is a restricted version of the “bucket” algorithm and we will therefore just discuss the “bucket” algorithm. All of the parameters for the algorithm reside in AdlerRouter and can be accessed via static get methods:

1. MARKING_BITS, Adler assumes this is 1, we use \( \log_2 16 = 4 \) bits.

2. TOTAL_BITS, Adler’s \( b - 1 \), this is the sum of the \( \log_2 \) of buckets to use and the marking bits. We default this to 8.

3. \( N \), Adler’s \( n \), the length of the longest path we will trace, we set it to 16.

4. RHO, Adler’s \( \rho \), we set it to \( 1/n = 1/16 \) as described in the paper.

5. DELTA, Adler’s \( \Delta \), this determines the number of packets we need to receive to compute a trace with probability \( 1 - \Delta \), we set this to 0.05.

6. EPSILON, Adler’s \( \epsilon \), used to determine the probability that a router resets a packet in the non-“bucket” version of the algorithm, we set this to 0.01.

AdlerRouter.java overrides Node.java’s run() method. We did this so we could see which interface each packet arrives on. In particular, when the AdlerRouter is initialized, it generates a hashtable to map each IP_Filter in the node’s in_filter table to an integer between 0 and 15 inclusive. When we receive a packet in our run() method we look at the filter that received it and then call a modified forward() method that takes the interface number as an additional argument. In forward() we mark the packet according to several parameters: with probability RHO we set the counter bits (we use the ipid field in Project_IP_Packet) to the binary representation of 1 and then with probability .5 we mark the interface (the tos in Project_IP_Packet) that was provided by forward() and with probability .5, clear the counter to 0s. With probability \( 1 - \text{RHO} \) we increment the counter and if the resulting value is 1 we set the packet’s interface number to the interface it arrived on with probability .5 and with probability .5 forward the interface as is.

AdlerRouter also has a thread that listens to the UDP port bound at 44444. This thread handles requests for mappings between interface numbers and IP addresses. Upon receiving such a request, the thread does a lookup based on the supplied interface number and returns the IP address at the other end of the interface. This is used during the second half of path reconstruction as discussed below.

AdlerTraceback is a slightly more complicated class. The most important data structures in this class are totals and sums. totals is a one-dimensional array that tracks the total number of packets received with a counter set to \( i \) in totals[i]. sums is a two-dimensional array that tracks the sums for each marking bit. sums is first indexed by which marking bit is being summed and then on which bucket a packet belongs to. inBPF() analyzes packets and increments the necessary positions in the arrays. For example, if a packet arrived with the counter set to 9 and marking bits in the form 1011, then inBPF would increment totals[9], sums[3][9], sums[1][9] and sums[0][9].

The most important methods in AdlerTraceback are trace, determineInterfaces() and decode(). trace is the call made from Simnet. It first determines the interfaces at each distance via a call to determineInterfaces(), then converts the resulting binary array back to decimal values. It then makes a call to traceNodes, which contacts each router based with the corresponding interface number and requests the IP address at the other end of the interface. This step is the actual act of traceback, we do not consider it part of the computation that occurs at the victim.
**determineInterfaces** is the most complex function. We will only discuss the part of it that is executed in the “bucket” algorithm because the other case is a subset of this case. This method is executed in nested loops. We keep a local two-dimensional array A. This as indexed as **sums**, except its second field is of length \( N + 1 \) (we ignore \( A[1][0] \)). When we are done, \( A[0] \) will hold the binary expansion associated with the probability for the first bit, \( A[1] \) for the second and so forth. For each bit we are tracking, i.e., this first field in **sums** we compute the “bucket” algorithm as if we only had one bit. The “bucket” algorithm works as follows. For each bucket (the second field in **sums**) we compute an array \( c \) (that Adler calls \( c^k_j \)). This array will hold roughly incrementing powers of \( .5 \) and will be used to expand our probability into binary with some error correction for probabilistic short-comings. This array will rely on two parameters \( q_{barn}^k \) (Adler’s \( q^n_k \)) and \( \alpha_{jk} \) (Adler’s \( \alpha^k_j \)). The former parameter is an estimate of the percent of the total flow that arrives at the victim with the counter set to \( k \). The latter measures the probability that the packet is reset within a certain distance of the victim. \( q_{barn} \) is computed slightly differently than is indicated in the paper. In the paper, this parameter is based on the total number of packets received, \( F \). However, we rarely wait for this many packets, timing constrains makes this infeasible. Thus, we change the computation of \( q_{barn} \) to be based on the total number of packets we received. \( \alpha_{jk} \) is computed as described in the paper.

After we have determined these parameters and the additional error correction parameter \( \sigma_k \) (Adler’s \( \sigma_k \)) we call **decode()**. This method computes the binary expansion and returns an array of 1s and 0s. Each call to **decode** will return a piece of the final interfaces array (A). For example, if we are working with the 0th marking bit and we are examining the first bucket, there are 4 buckets and an attack path of length of 16, then **decode()** will return values that will go into \( A[0][1], A[0][5], A[0][9] \) and \( A[0][13] \). The values for the second bucket will go into \( A[0][2], A[0][6], A[0][10] \) and \( A[0][14] \), etc.. Once this is done, **trace**() will use the resulting dual-subscripted array to to resolve the binary arrays into one decimal array and will consequently perform the actual traceback.

### 3 Topologies

Our original proposal stated that we would examine the different schemes under different topologies. Although we found that we had plenty of work without analyzing different topologies, we still needed a way to generate large topologies to run our tests. Under advisement, we decided to try to find a way to convert topologies generated by Boston university’s Representative Internet Toplogy gEnerator (BRITE) [5]. We implemented the **BriteToSimnet** class that would take topologies generated by the BRITE generator and create topologies for Simnet. Basically, this class takes a BRITE output file, which only specifies routers and links, and randomly assigns IP addresses to the routers and generates hosts. Each router has 16 fixed bits. Writing this class proved to be a rather tedious process because BRITE output changes slightly based on whether you are using the C++ or Java engine. Our class only works with the Java engine. Furthermore, the output format changes slightly depending on which generation methodology you are using (AS, Router, Top-down or Bottom-up), we believe our class resolves these discrepancies.

**Simple10.net** is a small, highly-connected network of ten routers and six hosts (Figure 1). There are no bandwidth limits or latencies on any of the links, as this network was designed to measure router overhead. Imposing bandwidth limits or latencies would have introduced an extra variable into the overhead calculations and would not have been an acceptable design element. The smaller radius of this network allowed us to accomplish the number of iterations we desired in a reasonable time frame.
Figure 1: Simple network topology generated by BRITE/BriteToSimnet that is used for testing router overhead.

Although BRITE specifies different parameters that can be used to change connectivity, we did not have much luck reducing connectivity without generating a very large network. As a result, the network that we used for testing attacks was large. This had the distinct disadvantage that loading it took quite a while. As a result, many of our test scripts actually generated large Simnet scripts that repeated the same commands and only loaded the topology once. The network that we used for most of our tests was named attack_network.net. attack_network consists of 75 routers, 38 hosts and 138 bi-directional links. Most of our tests examined an attack that traversed 10 hops to the victim H1, and thus came from H9, H10, H14, H16, H17, H21 and H28. We did not change the bandwidth, loss rate, etc. because we wanted to have tight control over the parameters we were measuring. If we were sending a SYN flood of 100,000 packets, we wanted all packets to get there. Furthermore, tests were very time consuming, so introducing latency and limiting bandwidth only hurt us.

4 Results

4.1 Path Reconstruction Metrics

This section discusses the resources required to generate traces for each algorithm. We discuss the processing required by the end host and the number of packets needed to perform an accurate trace.

4.1.1 Victim Reconstruction Time

We measured the reconstruction time for each algorithm by flooding a victim from a distance of 10 hops, waiting for the entire flood to arrive\footnote{Each of these algorithms require some sort of communication with upstream routers. Such communication would generally take place during the actual flood. Because we wait until our flood subsides (so we can be sure of how many packets we are processing), these tests do not take delay associated with a heavy flood during calculations into account.} and then measuring the time it took for the victim to generate a
trace. These tests were performed on a 1.25 GHz G4 processor, with 768 MB of RAM running Java 1.4.1 on Mac OS X.3. There were no page-outs during the test, so we assume that our code fit into memory and that there were no inconsistencies due to paging. We ran this test overnight using a shell script, so the only user processes executing on the system were `grep`, `cut` and `Simnet` We flooded with 100, 500, 1,000, 5,000, and 10,000 packets\(^2\). Our results are summarized in Figure 2 and Table 1.

![Traceback Overhead](image)

**Figure 2:** Relative traceback overhead between different algorithms. Measured in milliseconds, 100 floods.

<table>
<thead>
<tr>
<th></th>
<th>Adler Traceback</th>
<th>Ballard/Kimball Traceback</th>
<th>Song Traceback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>15.77</td>
<td>105</td>
<td>9094.19</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>2.84</td>
<td>23.49</td>
<td>467.55</td>
</tr>
<tr>
<td>Median</td>
<td>14</td>
<td>110</td>
<td>9068</td>
</tr>
<tr>
<td>Upper 25% ≥</td>
<td>15</td>
<td>119</td>
<td>9279</td>
</tr>
<tr>
<td>Lower 25% ≤</td>
<td>14</td>
<td>98</td>
<td>8743</td>
</tr>
</tbody>
</table>

**Table 1:** Relative traceback overhead between different algorithms, measured in milliseconds. (10,000 packets, 100 floods).

These tests measure the time needed to process enough information to be able to determine the attack path. The results are just as we predicted. Adler requires little time to determine interface numbers because his reconstruction algorithm consists of several exponentiations and converting decimal probabilities to their binary representations with error correction. Because the number of calculations is independent of

\(^2\)Note, at smaller floods our traces were not necessarily accurate. Due to the overhead generated by Song, we did not have time to perform statistically-significant tests with larger floods. However, these tests do serve the purpose of illustrating the difference between the algorithms.
the number of packets received, the time is constant (it is, however, a function of the number of bits used in the algorithm). Adler would further suffer from communicating with upstream routers during floods (after his computation has been completed). However, this communication can be done at a later time and is not necessary for the actual processing of data.

Our algorithm required more time than Adler’s at higher numbers of packets because we must perform graph-reconstruction. In the case of a single attacker, this is not extremely expensive, but when there are multiple attack paths, this overhead can grow exponentially. Furthermore, we must authenticate each packet we receive. Thus, the computation required for our algorithm grows with the number of packets, as is apparent from Figure 2. This algorithm would also suffer from having to communicate with upstream routers during sustained floods. However, unlike Adler, this communication MUST be performed before any processing takes place.

Song’s algorithm is by far the most expensive in terms of end-host processing. The reason Song is so expensive is that it must first determine the network topology before it can start to perform reconstruction. This is due to the small number of bits used in the algorithm, the end-host does not know which router is marking the packets it receives. Our algorithm does not suffer from this because we learn information about our topology from our packets. Song’s algorithm suffers from the same two drawbacks as ours: 1, it takes longer with the number of paths that it must compute and 2, its performance is a function of the number of packets it is processing. In addition to this fact, Song must perform two authentications per packet, one for the original marking router and one for the second.

### 4.1.2 Packets Required for Accurate Traces

We also performed tests to determine the number of packets required by each algorithm to compute an accurate trace of 10 hops. For each algorithm, we flooded with 100, 500, 1,000, 5,000, 10,000, 50,000 and 100,000 packets. We repeated these tests 500 times for the first five numbers and 250 times for the second two (we had to do this due to the amount of time it took Simnet to process the packets and Song to compute the traces). We used a utility called **PacketWaiter** to guarantee that we did not start reconstruction until we received the entire flood. We found that if we did not do this, and simply switched between nodes after a flood we sometimes started the reconstruction before the flood had subsided. This was bad because we were trying to measure the number of packets required for an accurate trace. Our results are summarized in Figure 3, and are what we expected.
Figure 3: Packets Required for accurate traces. (500 floods for less than 50,000 packets, 250 for ≥ 50,000 packets)

Song required the least number of packets, achieving 100% accuracy at 5000 packets (and 99% accuracy at 2500 packets). Basically, Song only has to receive one authenticated packet for each router along the attack path. If we were able to authenticate every single packet, we would be able to bound the number of packets required for an accurate trace as $\frac{\ln d}{p(1-p)^{d-1}}$ (where $d$ is the distance and $p$ is the marking rate) as described in Savage [6]. We found that we achieved the best results at a marking rate of 25% (although Savage suggests $1/d$ would be optimal, we will come back to this shortly). Thus at a distance of 10, we should have needed about 122 packets. The reason this number was not attained is that we do not just have receive packets for Song, we must also authenticate them. Unfortunately, it was rather difficult to determine the correct time slice and $\delta$ (which is to take into account TimerTask inconsistencies and the time it takes for a packet to get from the Router to the victim) to allow accurate authentication. To authenticate a packet we check for the time slice $\pm \delta$. We found that, in general, if we set our time slice to 1 second, the routers would actually publish around every 1007.93 ms (with a standard deviation of 9.689 ms). We found we attained our best results when $\delta = 250$ ms for Song. Unfortunately, we also found that this varied between the two systems we were testing on. Also, because we compute each key for previous time slices based on the S/Key algorithm, keys published earlier are less likely to be correlated correctly. For example, once we have a key $K_t$ for time $t$, we compute $K_{t-1}$ for time $t-1$ and so forth. At the victim, time $t-1$ will be exactly one second less that time $t$, but at the Routers it could be 1007 ms earlier. For $K_t$, our generated keys should be within the time slice $\pm \delta$ range, but as we compute keys for times that are much earlier, the addition of all of the TimerTask delays will add up and the times computed by S/Key will eventually become unsynchronized with the actual published times.

Our algorithm required the next highest number of packets, achieving 100% accuracy by 50,000 packets. This large number is due to our tiny marking rate of 0.125%. If we followed the bound described in [6] and were able to authenticate every packet, we would need approximately 1,862 packets. Again, our key publishing algorithm suffered the same difficulties as Song. However, we were able to authenticate more
packets. Song could fail to authenticate a packet if either of the hashes did not match up (by the first marking router and the second). However, we only compute one authentication hash so we have less chance for a miss.

As expected, Adler required the largest number of packets (except in a certain case, which will be discussed in §4.4). What is interesting is that even at 100 packets, his scheme did work about 4% of the time. This is not due to the random nature of interface numbers, because we flooded along multiple paths to eliminate such a problem. For our parameters ($\Delta = .05$, $\rho = 0.0625$, 4 counter bits and $n = 16$) we achieved approximately 76.95% accuracy at 100,000 packets. Given these parameters, Adler stated that we would require $O\left(\frac{2^{2n^2}}{\pi n^2} \ln(2^b/\Delta)\right)$ packets, which is 211729c packets, for some constant $c$, to compute a trace with 95% probability (in our case, $b = 5$). Due to timing considerations, and the unknown value $c$, we were not able to perform heavy testing at this number, but for a small number of floods of 200,000 packets (25) we received 100% accuracy.

### 4.2 Router Overhead

We measured the router overhead in terms of the time it took for each of the six hosts in our simple10.net network to flood each of the other five hosts on the network with 1,000 packets. After 250 iterations, we established the mean, median, standard deviation, and upper and lower quartiles for the routers.

Each packet sent by the flooder resulted in two packets traversing the network, since each SYN packet sent by the attacker resulted in a RST being sent by the victim. A flood of 1,000 packets to each of five hosts therefore resulted in 10,000 packets traversing the network. Since this was done by six hosts in turn, 60,000 packets were sent on each iteration. The throughput measured is the number of seconds it took for every packet in this flood to go from the attacker to the victim. The PacketWaiter class was implemented to ensure that every packet sent in the flood arrived at its destination. This class blocked execution of our script so that our elapsed-time calculations could only occur after all packets had been received by the victim. The throughput was measured on an Atholon 1600+ system with 768MB of RAM, running Sun’s j2sdk1.4.2 on Mandrake.

We established a baseline performance figure using simnet.Router. The mean transfer time for this router was 18,252ms, with a standard deviation of 53ms. The median was 18,250ms and the upper quartile was greater than 18,276, with a lower quartile less than 18,223ms (Table 2). The observed throughput was therefore 3,287 packets per second.

We fully expected SongRouter to suffer from significant router overhead because it must generate a new hash for each packet that it marks. This was measured as a 37.49 percent increase over the default simnet.Router transfer time. The mean transfer time for a six-flood round was 25,093ms, with a standard deviation of 85ms. The median was 25,095ms and the upper quartile was at 25,151ms with a lower quartile of 25042ms (Table 2). The observed throughput was therefore 2,391 packets per second.

We did not expect AdlerRouter to suffer from significant router overhead, because it did not have to compute any keys or hashes. The only significant additions to the router are a coin-flip and a linking of an interface identification number to an interface (which is done via a hashtable). This resulted in a 2.73 percent increase over the default simnet.Router transfer time, which was fully within our expectations. The mean transfer time for a six-flood round was 18,750ms, with a standard deviation of 58ms. The median was 18,748ms and the upper quartile was at 18,778ms with a lower quartile of 18,718ms. (Table 2.) The observed throughput was therefore 3,200 packets per second.

We expected BallardKimballRouter to have a router overhead greater than that of simnet.Router and somewhere between that of AdlerRouter and that of SongRouter. Our results were consistent with this expectation, showing a 3.22 percent increase over the default simnet.Router. BallardKimballRouter demonstrated transfer times 0.48 percent greater than that of AdlerRouter, but 24.92 percent lower than that of SongRouter. The mean transfer time for a six-flood round was 18,839ms, with a standard deviation
of 449ms. The median was 18,781ms and the upper quartile was at 18,816ms, with a lower quartile of 18739ms (Table 2). The observed throughput was therefore 3,185 packets per second.

The router performances were consistent with our expectations and show that of the traceback-enabled routers, AdlerRouter imposed the least overhead. Not surprisingly BallardKimballRouter imposed less overhead than SongRouter because it does not have to calculate new hashes for each packet (Figure 4.)

![Figure 4: Relative Router overhead between different router classes. Measured in milliseconds to receive 60,000 packets, 250 iterations.](image)

<table>
<thead>
<tr>
<th>Router</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Median</th>
<th>Upper 25%</th>
<th>Lower 25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simnet</td>
<td>18250</td>
<td>53.42</td>
<td>18223</td>
<td>&gt;18276</td>
<td>&lt;18718</td>
</tr>
<tr>
<td>Adler</td>
<td>18749.92</td>
<td>57.62</td>
<td>18718.0</td>
<td>&gt;18778</td>
<td>&lt;18739</td>
</tr>
<tr>
<td>BallardKimball</td>
<td>18839.18</td>
<td>448.77</td>
<td>18781.0</td>
<td>&gt;18816</td>
<td>&lt;18739</td>
</tr>
<tr>
<td>Song</td>
<td>25093.26</td>
<td>85.00</td>
<td>25095.0</td>
<td>&gt;25151</td>
<td>&lt;25042</td>
</tr>
</tbody>
</table>

Table 2: Relative Router overhead between different router classes. Measured in milliseconds to receive 60,000 packets, 250 iterations.

### 4.3 Impact on Regular Traffic

Since IP packets do not by default have room for the marking information needed by all of these schemes, they must be modified by these schemes in ways which contradict their intended uses. The algorithm defined by Song[8] overrides the identification field of the IP packet, which is used in fragment reconstruction. Any IP fragment that is marked is therefore lost, though according to [3] less than 0.25 percent of internet traffic is fragmented and would be subject to loss under this marking scheme. The algorithm defined by
Adler[1] does not specify where the marking is to take place, but in our AdlerRouter we mark in the TOS field and the IPID field, which [3] has shown are available for our use. Changes to the TOS field make no difference in packet delivery.

Our algorithm obviously has the most impact on network traffic because it essentially causes the loss of the packet which it is marking. Our extremely low marking rate is designed to minimize this impact. In order to gauge the impact of various marking rates on TCP traffic, we setup a simnet.TCPEchoClient and a simnet.TCPEchoServer on the same network we used for our path reconstruction tests. We then attempted to establish a connection between the attacker and the victim, with the server on the victim used in the path reconstruction tests and the client on one of the attackers used in the path reconstruction tests. If a connection could be established, we sent an echo request and marked the test as passed only if the client received a valid echo.

Using this metric, we found that with our proposed marking rate of 0.125 percent, we were able to complete 94.4 percent of our echo attempts. Losses at other marking rates are shown in (Table 3). We realize that even a rate of ninety-five percent is unacceptable in return for the added value of our marking scheme. (Table 5).

![Impact of Ballard/Kimball marking on TCP connections](image)

**Figure 5:** The Impact of Ballard/Kimball on TCP connections. (10 hop distance)
4.4 Robustness against Attacker-Marked Traffic

We will make two observations about the robustness of Song’s and Adler’s algorithms.

4.4.1 Adler

First, we tested the robustness of Adler’s algorithm by having PacketGenerator change the bits that we were using in our implementation of the algorithm (the tos and ipid fields). We tried four combinations where the tos field was either all 1s or all 0s and random and constant values for the counter. Our results are summarized in Figure 6. One would assume, and Adler states this observation in his paper, that the algorithm would perform the most accurately when the initial counter is randomized and the initial marking bits are set to 0. This is because if the counter is randomized, then we will get an even distribution of packets in the “buckets” at the victim. If the bits were set to 1 then it will take longer to attain the accurate probability distribution because any packet sent by the attacker that is not marked will increment each of the \( \log_2 L \) counters at the victim. If Adler’s reset policy truly works (and our implementation is correct), then the algorithm should withstand these modifications and perform similarly for each of the four combinations of initial conditions.

For our test we flooded along seven different paths of 10 hops. We flooded with 100, 500, 1,000, 5,000, 10,000, 50,000 and 100,000 packets and repeated the test 200 times for each combination. We used the same parameters as in §4.1.2. We found that the algorithm performed similarly when the marking bits were set to 0 for each of the counter variations and when the marking bits were set to 1 and the counter was constant. In these three cases, the results are as would be expected: when the counter is random and the bits are set to 0, we get about 82.57% accurate traces at 100,000 packets with a standard deviation of 10.3% between the 7 different attackers\(^3\) (Table 4). When the counter is constant and the marking bits are 0, we get 73.71% accurate traces at 100,000 packets with a standard deviation of 9.34% between attackers (Table 5). Finally, when the counter is constant and the marking bits are set to 1 we get 74.57% accuracy with a standard deviation of 6.70% between attackers (Table 6).

What is interesting is when the marking bits are set to 1 and the counter is random. Given the other three sets of traces, one would expect this variation to perform similarly. However, this combination achieved 99.57% accuracy with a standard deviation of 0.79% between attackers after only 500 packets! (Table 7) Each of the 1,400 traces was completely accurate at 1000 packets and greater. We are a little confused as to why this is happening, especially given the fact that the bits are set to 1. We were not able

---

\(^3\)This high standard deviation leads to the observation that attackers will have benefits based on their location. Due to the nature of the algorithm, this makes sense. Unfortunately, we did not have enough time to study this correlation further.
to come up with any viable hypotheses.\footnote{Please note that the results in §4.1 assume a smart attacker who is not setting the marking bits to 1 and using a random counter.}

Figure 6: The impact of changing the initial marking bits in the Adler Marking scheme. (10 hop distance) The number following ‘B’ is the initial setting of marking bits. ‘C:c’ indicates a constant counter value, ‘C:r’ indicates a random counter value.

<table>
<thead>
<tr>
<th>Packets:</th>
<th>100</th>
<th>500</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>50,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Accuracy:</td>
<td>3.86</td>
<td>4.29</td>
<td>4.71</td>
<td>11.14</td>
<td>20.29</td>
<td>56.86</td>
<td>82.57</td>
</tr>
<tr>
<td>Std.Dev:</td>
<td>2.19</td>
<td>2.93</td>
<td>2.43</td>
<td>5.49</td>
<td>6.58</td>
<td>7.10</td>
<td>10.37</td>
</tr>
<tr>
<td>Median:</td>
<td>5.00</td>
<td>4.00</td>
<td>4.00</td>
<td>9.00</td>
<td>18.00</td>
<td>56.00</td>
<td>84.00</td>
</tr>
<tr>
<td>Lowest quartile:</td>
<td>2.00</td>
<td>2.00</td>
<td>3.00</td>
<td>7.50</td>
<td>15.50</td>
<td>52.00</td>
<td>78.00</td>
</tr>
<tr>
<td>Highest quartile:</td>
<td>5.00</td>
<td>6.00</td>
<td>6.50</td>
<td>16.00</td>
<td>24.50</td>
<td>61.00</td>
<td>90.00</td>
</tr>
</tbody>
</table>

Table 4: Adler reconstruction: Marking bits 0, Counter Random

<table>
<thead>
<tr>
<th>Packets:</th>
<th>100</th>
<th>500</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>50,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Accuracy:</td>
<td>4.57</td>
<td>2.86</td>
<td>4.29</td>
<td>10.00</td>
<td>17.86</td>
<td>50.29</td>
<td>73.71</td>
</tr>
<tr>
<td>Std.Dev:</td>
<td>4.69</td>
<td>2.97</td>
<td>0.95</td>
<td>3.32</td>
<td>7.13</td>
<td>8.83</td>
<td>9.34</td>
</tr>
<tr>
<td>Median:</td>
<td>2.00</td>
<td>1.00</td>
<td>4.00</td>
<td>9.00</td>
<td>19.00</td>
<td>50.00</td>
<td>76.00</td>
</tr>
<tr>
<td>Lowest quartile:</td>
<td>1.50</td>
<td>1.00</td>
<td>4.00</td>
<td>8.00</td>
<td>12.50</td>
<td>43.00</td>
<td>68.00</td>
</tr>
<tr>
<td>Highest quartile:</td>
<td>6.00</td>
<td>5.00</td>
<td>4.50</td>
<td>11.50</td>
<td>21.50</td>
<td>56.00</td>
<td>80.00</td>
</tr>
</tbody>
</table>

Table 5: Adler reconstruction: Marking bits 0, Counter Constant
4.4.2 Song

Song’s authentication scheme has one major short-coming: it only uses an 11 bit hash for authentication. Goodrich noted that, because of this, there will be a hash collision every 2,048 packets because Song computes a new hash for each packet [4]. Due to this small number, we did not have to have PacketGenerator manipulate the bits the algorithm looked at; the sheer volume of traffic and our large number of routers did this for us. We found that if we flooded with 5,000 or more packets, that our implementation of Song’s algorithm would suffer from ‘splay.’ After a couple of hops, the path would fan out. What would happen is that the algorithm would have a packet that authenticated correctly for more than one pair of routers and would choose the incorrect pair. This happened rather infrequently, the splays were generally only caused by 1 or 2 packets and the correct path would still be one of the generated paths, but the inaccurate paths resulted in some false positives. For larger floods, the number of false paths was significant. Our algorithm did not suffer from this shortcoming because we used a 20 bit hash and we only authenticate on a per-router basis, not a per-packet basis. Given this fact, we would need to have a path that would traverse over 1,000,000 routers to suffer from the ‘splay’ phenomenon.

4.5 Out-of-band communication

All three of these probabilistic packet marking schemes rely on significant out of band communication. Both our scheme and that of Song [8] rely on the retrieval of keys from routers, while Adler relies on the ability to query a router for the address of the router connected to a specific link. In the case of our algorithm and Song’s, a twenty byte key plus an eight byte timestamp must be retrieved from each router at the time of path reconstruction. In addition, a signed seed-key must be published for every key chain the router generates and is used by the victim to validate the key chain. This key-size will be dependent on implementation and is not covered in our implementation. In the case of Adler, a 32-bit IP address must be returned to every querying victim, with the address corresponding to the router on the other end of the queried link.
In addition to communicating with routers, the Song algorithm must determine the network topology from the victim to every possible router. In SongTraceback we accomplish this by getting a list of all routers from simnet.Simnet and then using our specialized SongTraceroute to determine the distance of each router. SongTraceroute differs from simnet.Traceroute only in that it has no printout() statements and rather than simply printing out the distance and times it returns the distance value to SongTraceback.

5 Conclusion

Both our algorithm and that of Song [8] correctly identify paths to multiple attackers, however due to time constraints we were unable to test these assumptions extensively. Adler’s [1] algorithm does not work for multiple attackers without further significant research, as Adler claims the algorithm can be extended to address multiple attackers, though he provides no solution to this problem.

Our algorithm offers significant performance improvements over the Song algorithm because of the reduced likelihood of hash collisions as well as the reduced reconstruction times and router overhead. By incurring low reconstruction and router overhead and addressing multiple attackers, we feel that of the three algorithms implemented, ours represents a solid ratio of performance to overhead. Unfortunately, our algorithm does induce losses and therefore is not an acceptable solution. However, we feel that this project has demonstrated the need to use more bits to perform accurate, efficient, authenticated traceback.

References


