Detecting Stepping Stones

Yin Zhang and Vern Paxson*

Abstract

One widely-used technique by which network attackers attain anonymity and complicate their apprehension is by employing *stepping stones*: they launch attacks not from their own computer but from intermediary hosts that they previously compromised. We develop an efficient algorithm for detecting stepping stones by monitoring a site's Internet access link. The algorithm is based on the distinctive characteristics (packet size, timing) of interactive traffic, and not on connection contents, and hence can be used to find stepping stones even when the traffic is encrypted. We evaluate the algorithm on large Internet access traces and find that it performs quite well. However, the success of the algorithm is tempered by the discovery that large sites have many users who routinely traverse stepping stones for a variety of legitimate reasons. Hence, stepping-stone detection also requires a significant policy component for separating allowable stepping-stone pairs from surreptitious access.

1 Introduction

A major problem with apprehending Internet attackers is the ease with which attackers can hide their identity. Consequently, attackers run little risk of detection. One widely-used technique for attaining anonymity is for an attacker to use *stepping stones*: launching attacks not from their own computer but from intermediary hosts that they previously compromised. Intruders often assemble a collection of accounts on compromised hosts, and then when conducting a new attack they log-in through a series of these hosts before finally assaulting the target. Since stepping stones are generally heterogeneous, diversely-administered hosts, it is very difficult to trace an attack back through them to its actual origin.

There are a number of benefits to detecting stepping stones: to flag suspicious activity; to maintain logs in case a break-in is subsequently detected as having come from the local site; to detect inside attackers laundering their connections through external hosts; to enforce policies regarding transit traffic; and to detect insecure combinations of legitimate connections, such as a clear-text Telnet session that exposes an SSH passphrase.

*Y. Zhang is with the Computer Science Department, Cornell University, Ithaca, NY. Email: yzhang@cs.cornell.edu. V. Paxson is with the AT&T Center for Internet Research at ICSI, at the International Computer Science Institute in Berkeley, CA, and with the Lawrence Berkeley National Laboratory. Email: vern@aciri.org. This paper appears in the Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, August 2000.

The problem of detecting stepping stones was first addressed in a ground-breaking paper by Staniford-Chen and Heberlein [SH95]. To our knowledge, other than that work, the topic has gone unaddressed in the literature. In this paper, we endeavor to systematically analyze the stepping stone detection problem and devise accurate and efficient detection algorithms. While, as with most forms of intrusion detection, with enough diligence attackers can generally evade detection [PN98], our ideal goal is to make it painfully difficult for them to do so.

The rest of the paper is organized as follows. We first examine the different tradeoffs that come up when designing a stepping stone algorithm (§ 3). We then in § 4 develop a timing-based algorithm that works surprisingly well, per the evaluation in § 5, and also evaluate two cheap context-based techniques. We conclude in § 6 with some of the remaining challenges: in particular, the need for rich monitoring policies, given our discovery that legitimate stepping stones are in fact very common; and the possibility of detecting non-interactive *relays* and *slaves*.

2 Terminology and Notation

We begin with terminology. When a person (or a program) logs into one computer, from there logs into another, and perhaps a number still more, we refer to the sequence of logins as a *connection chain* [SH95]. Any intermediate host on a connection chain is called a *stepping stone*. We call a pair of network connections a *stepping stone connection pair* if both connections are part of a connection chain.

Sometimes we will differentiate between *flow* and *connection*. A *bidirectional* connection consists of two *unidirectional* flows. We term the series of flows along each direction of a connection chain a *flow chain*.

We use the following additional notation:

- $h_1 \leftrightarrow h_2$: a bi-directional network connection between h_1 and h_2. We also use C_1, C_2, ... to denote network connections.
- $h_1 \rightarrow h_2$: a unidirectional flow from h_1 to h_2.
- $\equiv_{stepping}$ is a binary relation defined over all connections as follows: $C_1 \equiv_{stepping} C_2$ if and only if C_1 and C_2 form a stepping stone connection pair.

3 Design Space

In this section we discuss the tradeoffs of different high-level design considerations when devising algorithms to detect stepping stones. Some of the choices relate to the following observation about stepping-stone detection: intuitively, the difference between a stepping stone connection pair and a randomly picked pair of connections is that the connections in the stepping stone pair are much more likely to have some correlated traffic characteristics. Hence, a general approach for detecting stepping stones is to identify traffic characteristics that are *invariant* or at least highly correlated across stepping stone connection pairs, but not so for arbitrary pairs of connection. Some potential candidates for such invariants are the connection contents, inter-packet spacing, ON/OFF patterns of activity, traffic volume or rate, or specific combinations of these. We examine these as they arise in the subsequent discussion.

3.1 Whether to analyze connection contents

A natural approach for stepping-stone detection is to examine the contents of different connections to find those that are highly similar. Such an approach is adopted in [SH95] and proves effective. Considerable care must be taken, though, because we will not find a perfect match between two stepping stone connections. They may differ due to translations of characters such as escape sequences, or the varying presence of Telnet options [PR83b].

In addition, suppose we are monitoring connections $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$, where h_2 is the stepping stone the attacker is using to access h_3 from h_1. If we adopt a notion of "binning" in order to group activity into different time regions (for example to compute character frequencies as done in [SH95]) then due to the lag between activity on $h_1 \leftrightarrow h_2$ and activity on $h_2 \leftrightarrow h_3$, the contents falling into each bin will match imperfectly. Furthermore, if the attacker is concurrently attacking h_4 via h_2, then the traffic on $h_1 \leftrightarrow h_2$ will be a mixture of that from $h_2 \leftrightarrow h_3$ and that from $h_2 \leftrightarrow h_4$, and neither of the latter connections' contents will show up exactly in $h_1 \leftrightarrow h_2$.

These considerations complicate content-based detection techniques. A more fundamental limitation is that content-based techniques cannot, unfortunately, work when the content is encrypted, such as due to use of SecureShell (SSH; [YKSRL99]).

The goal of our work was to see how far we could get in detecting stepping stones without relying on packet contents, because by doing so we can potentially attain algorithms that are more robust. Not relying on packet contents also yields a potentially major performance advantage, which is that we then do not need to capture entire packet contents with the packet filter, but only packet headers, considerably reducing the packet capture load. However, we also devised two cheap content-based techniques for purposes of comparison (§ 5.3), neither of which is robust, but both of which have the virtue of being very simple.

3.2 Direct vs. indirect stepping stones

Suppose h_1, h_2, h_3 is a connection chain. The *direct* stepping stone detection problem is to detect that h_2 is a stepping stone if we are observing network traffic that includes the packets belonging to $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$. If, however, the connection chain is $h_1, h_2, \ldots, h_3, h_4$, then the *indirect* stepping stone detection problem is to detect that connections $h_1 \leftrightarrow h_2$ and $h_3 \leftrightarrow h_4$ form a stepping stone pair, given that we can observe their traffic but *not* the traffic belonging to $h_2 \ldots h_3$ (and hence there is no obvious connection between h_2 and h_3).

Detecting direct stepping stones can be simpler than detecting indirect ones because for direct ones we can often greatly reduce the number of candidates for connection pairs. On the other hand, it is much easier for attackers to elude direct stepping stone detection by simply introducing an additional hop in the stepping stone chain. Furthermore, if we can detect indirect stepping stones then we will have a considerably more flexible and robust algorithm, one which can, for example, be applied to traffic traces gathered at different places (see below).

In this paper we focus on the more general problem of detecting indirect stepping stones.

3.3 Real-time detection vs. off-line analysis

We would like to be able to detect stepping stones in real-time, so we can respond to their detection before the activity completes. Another advantage of real-time detection is that we don't have to store the data for all of the traffic, which can be voluminous. For instance, a day's worth of interactive traffic (Telnet/Rlogin) at the University of California in Berkeley on average comprises about 1 GB of storage for 20,000 connections.

Algorithms that only work using off-line analysis are still valuable, however, for situations in which retrospective detection is needed, such as when an attacked site contacts the site from which they were immediately attacked. This latter site could then consult its traffic logs and run an off-line stepping stone detection algorithm to determine from where the attacker came into their own site to launch the attack.

Since real-time algorithms generally can also be applied to off-line analysis, we focus here on the former.

3.4 Passive monitoring vs. active perturbation

Another design question is whether the monitor can only perform passive monitoring or if it can actively inject perturbing traffic to the network. Passive monitoring has the advantage that it doesn't generate additional traffic, and consequently can't disturb the normal operation of the network. On the other hand, an active monitor can be more powerful in detecting stepping stones: after the monitor finds a stepping-stone candidate, it could perturb one connection in the pair by inducing loss or delay, and then look to see whether the perturbation is echoed in the other connection. If so, then the connections are very likely correlated.

Here we focus on passive monitoring, both because of its operational simplicity, and because if we can detect stepping stones using only passive techniques, then we will have a more broadly applicable algorithm, one that works without requiring the ability to manipulate incidental traffic.

3.5 Single vs. multiple measurement points

Tracing traffic at multiple points could potentially provide more information about traffic characteristics. On the other hand, doing so complicates the problem of comparing the traffic traces, as now we must account for varying network delays and clock synchronization. In this paper, we confine ourselves to the single measurement point case, with our usual presumption being that that measurement point is on the access link between a site and the rest of the Internet.

3.6 Filtering

An important factor for the success of some forms of real-time stepping-stone detection is filtering. The more traffic that can be discarded on a per-packet basis due to patterns in the TCP/IP headers, the better, as this can greatly reduce the processing load on the monitor.

However, there is clearly a tradeoff between reduced system load and lost information. First, if a monitor detects suspicious activity in a filtered stream, often the filtering has removed sufficient accompanying context that it becomes quite difficult determining if the activity is indeed an attack. In addition, the existence of filtering criteria makes it easier for the attackers to evade detection by manipulating their traffic so that it no longer matches the filtering criteria. For example, an evasion against filtering based on packet size (see below) is to use a Telnet client modified to send a large number of do-nothing Telnet options along with each keystroke or line of input.

The main likely filtering criteria for stepping-stone detection is packet size. Keystroke packets are quite small. Even when entire lines of input are transferred using "line mode" [Bo90], packet payloads tend to be much smaller than those used for bulk-transfer protocols. Therefore, by filtering packets to only capture small packets, the monitor can significantly reduce its packet capture load (for example, by weeding out heavy bulk-transfer SSH sessions while keeping interactive ones).

3.7 Minimizing state for connection pairs

Since potentially there can be a large number of active connections seen by the monitor, it is often infeasible to keep stepping-stone state for all possible pairs of connections due to the N^2 memory requirements. Therefore we need mechanisms that allow us to only keep state for a small subset of the possible connection pairs.

One approach is to limit our analysis to only detecting direct stepping stones, but for the reasons discussed in § 3.2 above, this is unappealing. There are, however, other mechanisms that work well:

- Remove connection pairs sharing the same port on the same host. If $h_1 \leftrightarrow h_2$ and $h_2 \leftrightarrow h_3$ both use port p on host h_2, then most likely the two connections are merely using the same server on h_2, rather than h_1 accessing a server on h_2 and then from that server running a client on h_2 to access a server on h_3. Removing such connection pairs is particularly helpful when there are a large number of connections connecting to the same popular server—without such filtering, when k connections connect to the same server, we need to keep state for $\frac{k(k-1)}{2}$ connection pairs!

  Note that this mechanism is worth applying even if we also test for directionality (see below), because when the monitor analyzes already-existing connections, their directionality is not necessarily apparent.

- Remove connection pairs with inconsistent directions. Depending on the topology of the network monitoring point, we may be able to classify connections as "inbound" or "outbound." If so, then we can eliminate as connection pair candidates any pairs for which both connections are in the same direction. While these connections may in fact form a chain, if the monitoring location is a chokepoint, meaning the sole path into or out of the site, then in this case there will be another connection in the opposite direction with which we can pair either of these two connections. However, if the site has multiple ingress/egress points, then we can only safely apply such filtering if all such points are monitored and the monitors coordinate with one another.

- Remove connection pairs with inconsistent timing. If two connections are a stepping stone pair, then the "upstream" (closer to the attacker) connection should encompass the downstream connection: that is, it should start first and end last. Accordingly, we can remove from our analysis any connection pairs for which the connection that started earlier also terminates earlier.

  Note that there are two risks with this filtering. First, it may be that the upstream connection terminates slightly sooner than the downstream connection, because of details of how the different TCP shutdown handshakes occur. Second, this filtering may open up the monitor to evasion by an attacker who can force their upstream connection to terminate while leaving the downstream connection running.

3.8 Traffic patterns

We can coarsely classify network traffic as either exhibiting ON/OFF activity, or running fairly continuously. For the former, we can potentially exploit the traffic's timing structure (whether the ON/OFF patterns of two connections are similar). For the latter, we can potentially exploit traffic volume information (whether two connections flow at similar rates). In addition, even for continuous traffic, if the communication is reliable, any delays resulting from waiting to detect loss and retransmit may impose enough of an ON/OFF pattern on the traffic that we can again look for timing similarities between connections.

In this paper, we focus on traffic exhibiting ON/OFF patterns, as that is characteristic of interactive traffic, which arguably constitutes the most interesting class of stepping-stone activity.

3.9 Accuracy

As with intrusion detection in general, we face the problem of *false positives* (non-stepping-stone connections erroneously flagged as stepping stones) and *false negatives* (stepping stones the monitor fails to detect). The former can make the detection algorithm unusable, because it becomes impossible (or at least too tedious) to examine all of the alerts manually, and attackers can exploit the latter to evade the monitor.

In practice, the problem of comparing connections looking for similarities can be complicated by clock synchronization (if comparing measurements made by different monitors), propagation delays (the lag between traffic showing up on one connection and then appearing on the other), packet loss and retransmission, and packetization variations. Moreover, an intruder can intentionally inject noise in an attempt to evade the monitor. Therefore, the detection mechanism must be highly robust if it is to avoid excessive false negatives.

3.10 Responsiveness

Another important design parameter is the responsiveness of the detection algorithm. That is, after a stepping-stone connection starts, how long does it take for the monitor to detect it? Clearly, it is desirable to detect stepping stones as quickly as possible, to enable taking additional actions such as recording related traffic or shutting down the connection. However, in many cases waiting longer allows the monitor to gather more information and consequently it can detect stepping stones more accurately, resulting in a tradeoff of responsiveness versus accuracy.

Another consideration related to responsiveness concerns the system resources consumed by the detection algorithm. If we want to detect stepping stones quickly, then we must take care not to require more resources than the monitor can devote to detection over a short time period. On the other hand, if off-line analysis is sufficient, then we can use potentially more resource-intensive algorithms.

3.11 Open vs. evasive attackers

In general, intrusion detection becomes much more difficult when the attacker actively attempts to evade detection by the monitor [PN98, Pa98]. The difference between the two can come down to the utility of relying on heuristics rather than airtight algorithms: heuristics might work well for "open" (non-evasive) attackers, but completely fail in the face of an actively evasive attacker.

While ideally any detection algorithms we develop would be resistant to evasive attackers, ensuring such robustness can sometimes be exceedingly difficult, and we proceed here on the assumption that there is utility in "raising the bar" even when a detection algorithm can be defeated by a sufficiently aggressive attacker. In particular, for timing-based algorithms such as those we develop, we would like it to be the case that the only way to defeat the algorithm is for an attacker to have to introduce large delays in their interactive sessions, so that their inconvenience is maximized. We assess our algorithm's resistance to evasion in § 4.4.

4 A Timing-Based Algorithm

In this section we develop a stepping-stone detection algorithm that works by correlating different connections based solely on timing information. As discussed in the previous section, our design is motivated in high-level terms by the basic approach of identifying invariants. Moreover, the algorithm leverages the particulars of how interactive traffic behaves. This leads to an algorithm that is very effective for detecting interactive traffic (see evaluation in § 5), and should work well for detecting other forms of traffic that exhibit clear ON/OFF patterns.

4.1 ON/OFF periods

We begin by defining ON and OFF periods. When there is no data traffic on a flow for more than T_{idle} seconds, the connection is considered to be in an *OFF period*. We consider a packet as containing data only if it carries new (non-retransmitted, non-keepalive) data in its TCP payload. When a packet with non-empty payload then appears, the flow ends its OFF period and begins an *ON period*, which lasts until the flow again goes data-idle for T_{idle} seconds.

The motivation for considering traffic as structured into ON and OFF periods comes from the strikingly distinct distribution of the spacing between user keystrokes. Studies of Internet traffic have found that keystroke interarrivals are very well described by a Pareto distribution with fixed parameters [DJCME92, PF95]. The parameters are such that the distribution exhibits *infinite variance*, which in practical terms means a very wide range of values. In particular, large values are not uncommon: about 25% of keystroke packets come 500 msec or more apart, and 15% come 1 sec or more apart (1.6% come 10 sec or more apart). Thus, interactive traffic will often have significant OFF times. We can then exploit the tendency of machine-driven, non-interactive traffic to send packets back-to-back, with a very short interval between them, to discriminate non-interactive traffic from interactive.

4.2 Timing correlation when OFF periods end

The strategy underlying the algorithm is to correlate connections based on coincidences in when connection OFF periods end, or, equivalently, when ON periods begin.

Intuitively, given two connections C_1 and C_2, if $C_1 \equiv_{stepping} C_2$, it is very likely that C_1 and C_2 *often* leave OFF periods at *similar* times—the user presses a keystroke and it is sent along first C_1 and then shortly along C_2, or a program they have executed finishes running and produces output or they receive a new shell prompt (in which case the activity ripples from C_2 to C_1).

The inverse is also likely to be true. That is, if C_1 and C_2 often leave OFF periods at similar times, then it is likely that $C_1 \equiv_{stepping} C_2$, because there are not many other mechanisms that can lead to such coincidences. (We discuss two such mechanisms in § 5.7: periodic traffic with slightly different periods, and broadcast messages.)

By quantifying *similar* and *often*, we transform the above strategy into the following detection criteria:

1. We consider two OFF periods *correlated* if their ending times differ by $\leq \delta$, where δ is a control parameter.

2. For two connections C_1 and C_2, let OFF_1 and OFF_2 be the number of OFF periods in each, and $\text{OFF}_{1,2}$ be the number of these which are correlated. We then consider C_1 and C_2 a stepping stone connection pair if:

$$\frac{\text{OFF}_{1,2}}{\min(\text{OFF}_1, \text{OFF}_2)} \geq \gamma,$$

   where γ is a control parameter, which we set to 0.3.

A benefit of this approach is that the work is done *only after significant idle periods*. For busy, non-idle connections (far and away the bulk of traffic), we do nothing other than note that they are still not idle. Related to this, we need consider only a small number of possible connection pairs at any given time, because we can ignore both those that are active and those that are idle; we need only look at those that have transitioned from idle to active, and that can't happen very often because it first requires the connection to be inactive for a significant period of time. Consequently, the algorithm does not require much state to track stepping-stone pair candidates.

Because of the very wide range of keystroke interarrival times, the algorithm is not very sensitive to the choice of T_{idle}. In our current implementation, we set $T_{\text{idle}} = 0.5$ sec. In § 5.6 we briefly discuss the effects of using other values.

Finally, because we only consider correlations of when ON periods *begin*, rather than when they *end*, we are more robust to differences in throughput capacities. For two connections $C_1 \equiv_{stepping} C_2$, if C_1's throughput capacity is significantly smaller than C_2's, then an ON period on C_2 may end sooner than on C_1 (where the echo of the same data takes longer to finish transferring); but regardless of this effect, ON periods will *start* at nearly the same time.

4.3 Refinements

The scheme outlined above is appealing because of its simplicity, but it requires some refinements to improve its accuracy. The first of these is to exploit timing *casuality*, based on the following observation: if two flows F_1 and F_2 are on the same *flow chain*, then their timing correlation should have a consistent ordering. If we once observe that F_1 ends its OFF period before F_2, then it should be true that F_1 *always* ends its OFF period before F_2. Confining our analysis in this way weeds out many false pairs.

To further improve the accuracy of the algorithm, we use the number of *consecutive* coincidences in determining the frequency of coincidences, because we expect consecutive coincidences to be more likely for true stepping stones than for accidentally coinciding connections. More specifically, in addition to the test in § 4.2, to consider two connections C_1 and C_2 a stepping stone connection pair we require:

$$\text{OFF}^*_{1,2} \geq \min_{\text{csc}} \quad \text{and} \quad \frac{\text{OFF}^*_{1,2}}{\min(\text{OFF}_1, \text{OFF}_2)} \geq \gamma',$$

where $\text{OFF}^*_{1,2}$ is the number of consecutive coincidences, OFF_1 and OFF_2 are as before, and $\min_{\text{csc}}$ and γ' are new control parameters. We initially used only the first of these refinements, requiring either $\min_{\text{csc}} = 2$ or $\min_{\text{csc}} = 4$ consecutive coincidences, for direct or indirect stepping stones, respectively. This in general works very well, but we added the second requirement when we found that very long-lived connections could sometimes eventually generate consecutive coincidences just by chance. These can be eliminated by very low γ' thresholds; we use $\gamma' = 2\%$ and $\gamma' = 4\%$ for direct and indirect stepping stones, respectively.

4.4 Resistance to evasion

Since the heart of the timing algorithm is correlating idle periods in two different connections, an attacker can attempt to thwart the algorithm by avoiding introducing any idle times to correlate; introducing spurious idle times on one of the connections not reflected in the other connection; or stretching out the latency lag between the two connections to exceed δ.

To avoid connection idle times, it will likely not suffice for the attacker to simply resolve to type quickly. Given $T_{\text{idle}} = 0.5$ sec (§ 5.6), it just takes a slight pause to think, or delay by the server in generating responses to commands, to introduce an idle time.

A mechanical means such as establishing a steady stream of traffic on one of the connections but not on the other seems like a better tactic. If the intermediary and either upstream or downstream hosts run custom software, then doing so is easy, though this somewhat complicates the attacker's use of the intermediary, as now they must install a custom server on it. Another approach would be to use a mechanism already existing in the protocol between the upstream host and the intermediary to exchange traffic that the intermediary won't propagate to the downstream host; for example, an on-going series of Telnet option negotiations. However, as particular instances of such techniques become known, they may serve as easily-recognized *signatures* for stepping stone connections instead.

Even given the transmission of a steady stream of traffic, idle times might still appear, either accidentally, due to packet loss and retransmission lulls, or purposefully, by a site introducing occasional 500 msec delays into its interactive traffic to see whether a delay shows up in a connection besides the one deliberately perturbed. Such delays might prove difficult for an attacker to mask.

The attacker might instead attempt to introduce a large number of idle times on one connection but not on the other, so as to push the ratio of idle time coincidences below γ. This will also require running custom software on the intermediary, and, indeed, this approach and the previous one are in some sense the same, aiming to undermine the basis of the timing analysis. The natural counter to this evasion tactic is to lower γ, though this of course will require steps to limit or tolerate the ensuing additional false positives. It might also be possible to detect unusually large numbers of idle periods, though we have not characterized the patterns of multiple idle periods to assess the feasibility of doing so.

Another approach an attacker might take is to pick an intermediary for which the latency lag between the two connections is larger than δ, which we set to 80 msec in § 5.6. Doing so simply by exploiting the latency between the monitoring point and the intermediary is not likely to work well, as for most sites the latency between an internal host and a monitoring point will generally be well below 40 msec; however, if an internal host connected via a very slow link (such as a modem) is available, then that may serve. Another approach would be to run a customized server or client on the intermediary that explicitly inserts the lag of 80 msec. This approach appears a significant concern for the algorithm, and may require use of much larger values of δ, so as to render the delay highly inconvenient for the attacker (80 msec is hardly noticeable, much less inconvenient). This is a natural area for future work.

5 Performance Evaluation

In § 4 we developed a timing-based algorithm for stepping stone detection. We have implemented the algorithm in Bro, a real-time intrusion detection system [Pa98]. In this section, we evaluate its performance (in terms of false positives and false negatives) on traces of wide-area Internet traffic recorded at the DMZ access link between the global Internet and two large institutions, the Lawrence Berkeley National Laboratory (LBNL) and the University of California at Berkeley (UCB).

5.1 Traces used

We ran the timing-based algorithm on numerous Internet traces to evaluate its performance. Due to space limitations, here we confine our discussion to the results for two traces:

- `lbnl-telnet.trace` (120 MB, 1.5M packets, 3,831 connections): one day's worth of Telnet and Rlogin traffic collected at LBNL. (The traffic is more than 90% Telnet.)

- `ucb-telnet.trace` (390 MB, 5M packets, 7,319 connections): 5.5 hours' worth of Telnet and Rlogin traffic collected at UCB during the afternoon busy period.

The performance of the algorithm on other traces is comparable.

5.2 Brute force content-based algorithm

To accurately evaluate the algorithms, we first devised an off-line algorithm using brute-force content matching.

The principle behind the algorithm is that, for stepping stones, each line typed by the user is often echoed verbatim across the two connections (when the content is not encrypted). Therefore, by looking at lines in common, we can find connections with similar content. With additional manual inspection, we can identify the stepping stones.

The algorithm works as follows:

1. Extract the aggregate Telnet and Rlogin output (computer-side response), for all of the sessions in the trace, into a file.

2. For each different line in the output, count how many times it occurred (this is just *sort* | *uniq -c* in Unix).

3. Throw away all lines except those appearing exactly twice. The idea is that these are good candidates for stepping stones, in that they are lines unique to either one or at most two connections.

4. Find the connection(s) in which each of these lines appears. This is done by first building a single file listing every unique line in every connection along with the name of the connection, and then doing a database *join* operation between the lines in that file and those in the list remaining after the previous step.

   If a line appears in just one connection, throw the line away.

5. Count up how many of the only-seen-twice lines each pair of connections has in common (using the Unix *join* utility).

6. Connection pairs with 5 or more only-seen-twice lines in common are now candidates for being stepping stones.

7. Of those, discard the pair if both connections are in the same direction (both into the site or both out of the site).

8. Of the remainder, visually inspect them to see whether they are indeed stepping stones. Most are; a few are correlated due to common activities such as reading the same mail message or news article.

Clearly the methodology is not airtight, and it fails completely for encrypted traffic. But it provides a good baseline assessment of the presence of clear-text stepping stones, and detects them in a completely different way than the timing algorithm does, so it is suitable for calibration and performance evaluation.

For large traces, the requirement of 5 or more lines allows us to significantly reduce the number of connection pairs that we need to visually inspect in the end. This appears to be necessary in order to make the brute-force content matching feasible.

For small- to medium-sized traces, we also inspect the ones with 2, 3, or 4 lines in common. Sometimes we did indeed find stepping stones that were missed if we required 5 lines in common. But in most cases, these stepping stones were exceedingly short in terms of bytes transferred.

5.3 Simple content-based algorithms

For purposes of comparison, we devised two simple content-based algorithms. Both are based on the notion that if we can find text in an interactive login C_1 unique to that login, then if that text also occurs in C_2, then we have strong evidence that C_1 and C_2 are related.

The problem then is to find such instances of unique text. Clearly, virtually all login sessions are unique in some fashion, but the difficulty is to cheaply detect exactly how.

Our first scheme relies on the fact that some Telnet clients propagate the X-Windows `DISPLAY` environment variable [Al94] so that remote X commands can locate the user's X display server. The value of `DISPLAY` should therefore be unique, because it globally identifies a particular instance of hardware.

We modified Bro to associate with each active Telnet session the value of `DISPLAY` propagated by the Telnet environment option (if any), and to flag any session that propagates the same value as an already existing session. We find, however, that this method has little power. It turns out that `DISPLAY` is only rarely propagated in Telnet sessions, and, in addition, non-unique values (such as hostnames not fully qualified, or, worse, strings like "`localhost.localdomain:0.0`") are propagated.[1]

Our second scheme works considerably better. The observation is that often when a new interactive session begins, the login dialog includes a status line like:

[1]However, we *have* successfully used `DISPLAY` propagation to backtrace attackers, so recording it certainly has some utility.

```
    Last login: Fri Jun 18 12:56:58
          from hostx.y.z.com
```

The combination of the timestamp (which of course changes with each new login session) and the previous-access host (even if truncated, as occurs with some systems) leads to this line being frequently unique.

We modified Bro to search for the following regular expression in text sent from the server to the client:

```
    /^([Ll]ast +(successful)? *login)/ |
    /^Last interactive login/
```

We found one frequent instance of false positives. Some instances of the Finger service [Zi91] report such a "last login" as part of the user information they return. Thus, whenever two concurrent interactive sessions happened to finger the same user, they would be marked as a stepping stone pair. We were able to filter such instances out with a cheap test, however: it turns out that the Finger servers also terminate the status line with ASCII-1 ("control-A").

We refer to this scheme as "login tag", and compare its performance with that of the timing algorithm below. It works remarkably well considering its simplicity. Of course, it is not very robust, and fails completely for a large class of systems that do not generate status lines like the above, though perhaps for those a similar line can be found.

5.4 Accuracy

We first evaluate the accuracy of the algorithms in terms of their false negative ratio and false positive ratio. For `lbnl-telnet.trace`, we identified 23 stepping stone connection pairs among a total of 3,831 connections using the brute-force content matching as described above. (We inspected all connections with 2 or more lines in common, so 23 should be a very accurate estimation of the number of stepping stones.) One stepping stone is indirect (§ 3.2), the others were direct.

The timing-based detection algorithm reports 21 stepping stones, with no false positives and 2 false negatives. Both false negatives are quite short: one lasts for 15 seconds and the other lasts for 34 seconds.

For `ucb-telnet.trace`, due to the large volume of the data, for the brute-force technique we only inspected connections with 5 or more lines in common. We identified 47 stepping stones. In contrast, the timing-based algorithm detects 74 stepping stones. 5 out of the 47 stepping stones we identified using brute-force were missed by the timing algorithm. Among the 5 false negatives, 3 are very short either in terms of duration (less than 12 seconds) or in terms of the bytes typed (in one connection, the user logs in and immediately exits). We discuss the additional 32 stepping stones detected by the timing-based algorithm, but not by the brute-force technique, below.

To further assess performance, we ran both the "display" and the "login tag" schemes (§ 5.3) on ucb-telnet.trace. The "display" scheme reported 3 stepping stones, including one missed by the timing-based algorithm. "login tag" reported 20 stepping stones (plus one false positive, not further discussed here). Of these 20, the timing-based algorithm only missed one, which was exceedingly short—all the user did during the downstream session was to type exit to terminate the session. (This is also the stepping stone that was detected by the "display" algorithm but not by the timing algorithm.)

In summary, the timing-based algorithm has a low false negative ratio. To make sure that this does not come at the cost of a high false positive ratio, we visually inspected the additional 32 stepping stones reported by the timing-based algorithm for ucb-telnet.trace to see which were false positives.

It turns out that all of them were actual stepping stones. For example, there were a couple of stepping stones that used *ytalk*, a chat program. These fooled the brute-force content matching algorithm due to a lot of cursor motions. Another stepping stone fooled the content-matching approach because retransmitted data showed up in one of the transcripts but not the other.

Thus, we find that the timing-based algorithm is highly accurate in terms of both false positive ratio and false negative ratio, and works considerably better than the brute-force algorithm that we initially expected would be highly accurate.

5.5 Efficiency

The timing-based algorithm is fairly efficient. Under the current parameter settings, on a 400MHz Pentium II machine running FreeBSD 3.3, it takes 69 real-time seconds for lbnl-telnet.trace, and about 24 minutes for ucb-telnet.trace. The former clearly suffices for real-time detection. The latter, for a 5.5 hour trace, reflects about 10% of the CPU, and would appear likewise to suffice. Note that the relationship between the running time on the two traces is not linear in the number of packets or connections in the trace, because what instead matters is the number of *concurrent* connections, as these are what lead to overlapping ON/OFF periods that require further evaluation.

5.6 Impact of different control parameters

The proper choice of the control parameters is important for both the accuracy and the efficiency of the algorithm. We based the current choice of parameters on extensive experiments with various traffic traces, which we summarize in this section. With these settings, the algorithm performs very well in terms of both accuracy and speed across a wide range of network scenarios.

To assess the impact of the different control parameters, we systematically explored the portions of the parameter space on ucb-telnet.trace. Table 1 summarizes the different parameter settings we considered. Note that we keep the default settings for T_{idle} and γ' when exploring the parameter space, which we did to keep the size of the parameter space tractable. We chose to not vary these two parameters in particular because based on extensive experiments with various traffic traces, we have found that:

Parameter	Values
T_{idle} (sec)	0.5
δ (msec)	20, 40, 80, 120, 160
γ	15%, 30%, 45%
$\min_{\text{csc}}$	1, 2, 4, 8, 12, 16
γ'	2% for direct stepping stones; 4% for indirect stepping stones

Table 1: Settings for different control parameters.

- The algorithm is fairly insensitive to the choice of T_{idle}. This is largely because, as noted in § 4.1, human keystroke interarrivals are well described by a Pareto distribution with fixed parameters. The Pareto distribution has a distinctive "heavy-tail" property, i.e., pretty much no matter what value we choose for T_{idle}, we still have an appreciable number of keystrokes to work with. However, the larger the T_{idle}, the more likely that we will miss short stepping stones. The current choice of 0.5 sec is a reasonable compromise between exceeding most round-trip times (RTTs), yet maintaining responsiveness to short-lived connections.

- Although the current choices of γ' thresholds are very low, they suffice to eliminate those very long-lived connections that eventually generate consecutive coincidences just by chance, which is the only purpose for introducing γ'.

Finally, an important point is that the goal for this assessment is determining the best parameters to use for an unaware attacker. If the attacker actively attempts to *evade* detection, then as noted in § 4.4 alternative parameters may be required even though they work less well in general. The important problem of assessing how to optimize the algorithm for this latter environment remains for future work.

We ran the detection algorithm on ucb-telnet.trace for each of the 75 possible combinations of the control parameters and assessed the number of false positives and false negatives. For brevity, we only report the complete results for $\gamma = 30\%$, and briefly summarize the results for $\gamma = 15\%$ and $\gamma = 45\%$.

Table 2 gives the results for detecting direct stepping stones when $\gamma = 30\%$. We make four observations. First, the number of false positives is close to 0 for all combinations of δ and $\min_{\text{csc}}$ except for $\min_{\text{csc}} = 1$, which clearly is too lax. Second, the number of false negatives is minimized when $\min_{\text{csc}} = 2$, which is the default setting in the algorithm. Third, the choice of δ has little impact on the accuracy of the algorithm. Finally, the results for $\gamma = 15\%$ and $\gamma = 45\%$ (not

FP/FN (γ=30%)

δ (msec)	$\min_{\text{csc}}$					
	1	2	4	8	12	16
20	1/8	0/8	0/10	0/17	0/21	0/26
40	1/6	0/7	0/10	0/17	0/21	0/25
80	4/5	0/7	0/9	0/16	0/20	0/24
120	12/5	0/7	0/9	0/15	0/19	0/24
160	20/5	0/7	0/9	0/14	0/19	0/24

Table 2: Number of false positives (FP) and false negatives (FN) for detecting direct stepping stones when $\gamma = 30\%$.

shown) are highly similar to those for $\gamma = 30\%$, which means the algorithm is insensitive to the choice of γ.

We should also note two additional considerations regarding δ. First, it is sometimes necessary to use a relatively large δ, especially when the latency is high (for example, for connections that go through transcontinental or satellite links). High latency often means large variation in the delay, which can distort the keystroke timing characteristics. One possible solution to this problem would be to choose different δ's based on the RTT of a connection. This would also help with the latency-lag evasion technique discussed in § 4.4. But such adaptation complicates the algorithm, because estimating RTT based on measurements in the middle of a network path can be subtle, so we have left it for future study.

Second, large δ's also mean we must maintain state for more concurrent connection pairs, which can eat up memory and CPU cycles. Similarly, having a smaller T_{idle} means that we need to update state for connections more frequently, which in turn increases CPU consumption. To illustrate these effects, we increased δ from 80 msec to 200 msec and reduced T_{idle} from 0.5 sec to 0.3 sec. After this change, the time required to process `lbnl-telnet.trace` increases to 155 sec, more than double the 69 sec required with the current settings.

FP/FN (γ=30%)

δ (msec)	$\min_{\text{csc}}$					
	1	2	4	8	12	16
20	162/0	5/0	0/0	0/2	0/5	0/6
40	683/0	19/0	0/0	0/2	0/4	0/6
80	2,486/0	134/0	0/0	0/1	0/3	0/5
120	5,633/0	431/0	12/0	3/1	3/3	2/5
160	10,131/0	995/0	28/0	7/1	4/3	2/5

Table 3: Number of false positives (FP) and false negatives (FN) for detecting indirect stepping stones when $\gamma = 30\%$.

Table 3 summarizes the results for detecting indirect stepping stones when $\gamma = 30\%$. From the table it is evident that both the number of false positives and the number of false negatives are minimized when $\min_{\text{csc}} = 4$ and $\delta \leq 80$ msec. A smaller $\min_{\text{csc}}$ or a larger δ can significantly increase the number of false positives, while a larger $\min_{\text{csc}}$ can lead to more false negatives. When $\gamma = 45\%$, the number of false positives is in general smaller, but the optimal combination of $\min_{\text{csc}}$ and δ remains the same. When $\gamma = 15\%$, the number of false positives increases 150–300%, and for $\delta = 80$ msec and $\min_{\text{csc}} = 4$, increases from 0 false positives to 7.

These findings show that the current settings of the parameters are fairly optimal, at least for the `ucb-telnet.trace`, and that there is considerable room for varying the parameters in response to certain evasion threats (§ 4.4). We also note that there is no particular need to use the same values of T_{idle}, δ, and γ for both direct and indirect stepping stones, other than simplicity, and there may be room for some further performance improvement by allowing them to be specific to the type of stepping stone, just as for $\min_{\text{csc}}$ and γ'.

5.7 Failures

In this section we summarize the common scenarios that can cause the timing-based algorithm to fail. Some of these failures have already been solved in the current algorithm, but it is beneficial to discuss them, because they illustrate some of the subtleties involved in stepping stone detection.

- Excessively short stepping stones. In many cases, the timing-based algorithm missed a stepping stone simply because the connections were exceedingly short. In some cases, the "display" and "login tag" schemes are still able to catch these because both of them key off of text sent very early during a login session.

  On the other hand, often attackers can't do very much during such short stepping stones, so failing to detect them is not quite as serious as failing to detect longer-lived stepping stones.

- Message broadcast applications such as the Unix *talk* and *wall* utilities. Such utilities can cause correlations between flows because they cause the same text to be transmitted on multiple connections. However, these correlations will be of the form $h_1 \to h_2$, $h_1 \to h_3$; that is, the connection endpoint that breaks the idle period will be the same for both flows (h_1, in this case), whereas for a true stepping stone $h_1 \to h_2 \to h_3$ the endpoint breaking the idle period will differ (first h_1, then h_2). This observation led to the directionality criterion in § 3.7.

- Correlations due to phase drift in periodic traffic. Consider two connections C_1 and C_2 that transmit data with periodicities P_1 and P_2. If the periodicities are exactly the same, then the ON/OFF periods of the connections will remain exactly the same distance apart (equal to the phase offset for the periodicities). If, however, P_1 is slightly different from P_2, then the offset between the ON/OFF periods of the two will drift in phase, and occasionally the two will overlap. Such overlaps appear to be correlations, but actually are due to the periods being in fact *uncorrelated*, and hence able to drift with respect to one another.

This phenomenon is not idle speculation (see also [FJ94] for discussion of how it can lead to self-synchronization in coupled systems). For example, one of our traces includes two remote Telnet sessions to the same machine at the same time (involving different user IDs, but clearly the same user). The sessions had a period of overlap during which both sessions were running *pine* to check mail. For some reason, the *pine* display began periodically sending data in small chunks, with about a second between each chunk. These transmissions were initially out of sync, but sometimes sync'd up fairly closely. Before we added the rule on consecutive coincidences (parameters $\min_{\text{CSC}}$ and γ', discussed in § 4.3), these sessions had been reported as a stepping stone, because the ratio of coincidences was high enough. After we refined the algorithm, such spurious stepping stones went away (the rule on directionality discussed in the previous item would have also happened to succeed in eliminating this particular case).

- Large latency and its variation. As mentioned above, when a connection has a very high latency or large delay variation, we need to increase the value of δ (and, accordingly, γ, $\min_{\text{CSC}}$, and γ') in order to detect it. We have not yet modified the algorithm to do so because of complications in efficiently estimating a connection's RTT.

5.8 Experience with operational use

We initially expected that detecting a stepping stone would mean that with high probability we had found an intruder attempting to disguise their location. As the figures above on the frequency of detecting stepping stones indicate, this expectation was woefully optimistic. In fact, we find that wide-area Internet traffic abounds with stepping stones, virtually all of them legitimate.

For example, UCB's wide area traffic includes more than 100 stepping stones each day. These fall into a number of categories. Some are external users who wish to access particular machines that apparently trust internal UCB hosts but do not trust arbitrary external hosts. Some appear to reflect habitual patterns of use, such as "to get to a new host, type *rlogin* to the current host," in which it is not infrequent to observe a stepping stone using a remote host to access a nearby local host, or even the same local host.[2] Some are simply bizarre, such as one user who regularly logs in from UCB to a site in Asia and then returns from the Asian site back to UCB, incurring hundreds of msecs of latency (and thwarting our default choice of δ, per the above discussion). Other possible legitimate uses that we haven't happened to specifically identify are gaining anonymity for purposes other than attacks, or running particular client software provided by the intermediary but not by the upstream host.

[2] Inspection of some of these connections confirms that these are not inside attackers attempting to hide their location.

Clearly, operational use will require development of refined Bro policy scripts to codify patterns corresponding to legitimate stepping stones, allowing the monitor to then alert only on those stepping stones at odds with the policies. But even given these hurdles, we find the utility of the algorithm clear and compelling.

Finally, we note that the detection capability has already yielded an unanticipated security bonus. Since the timing algorithm is indifferent to connection contents, it can readily detect stepping stones in which the upstream connection is made using a clear-text protocol such as Telnet or Rlogin, but the downstream connection uses a secure, encrypted protocol such as SSH. Whenever we detect such stepping stones, it is highly probable that the user typed their SSH passphrase or password in the clear over the first connection in the chain, thus undermining the security of the SSH connection. Indeed, after beginning to run the timing algorithm to look for this pattern, we rapidly found instances of such use, and confirmed that for each the passphrase was indeed typed in the clear. At LBNL, running the timing algorithm looking for such exposures is now part of the operational security policy, and, unfortunately, it continues to alert numerous times each day (and we have traced at least one break-in to a passphrase exposed in this manner at another site). Efforts are being made to educate the users about the nature of this risk.

6 Concluding remarks

Internet attackers often mask their identity by launching attacks not from their own computer, but from an intermediary host that they previously compromised, i.e., a stepping stone. By leveraging the distinct properties of interactive network traffic (smaller packet sizes, longer idle periods than machine-generated traffic), we have devised a stepping-stone detection algorithm based on correlating the timing of the ON/OFF periods of different connections. The algorithm runs on a site's Internet access link. It proves highly accurate, and has the major advantage of ignoring the data contents of the connections, which means both that it works for encrypted traffic such as SSH, and that the packet capture load is greatly diminished since the packet filter need only record packet headers.

While the algorithm works very well, a major stumbling block we failed to anticipate is the large number of legitimate stepping stones that users routinely traverse for a variety of reasons. One large site (the University of California at Berkeley) has more than 100 such stepping stones each day. Accordingly, the next step for our work is to undertake operating the algorithm as part of a site's production security monitoring, which we anticipate will require refined security policies addressing the many legitimate stepping stones. But even given these hurdles, we find the utility of the algorithm clear and compelling.

Finally, a natural extension to this work is to attempt to likewise detect non-interactive stepping stones, such as *relays*, in which traffic such as Internet Relay Chat [OR93] is looped through a site, and *slaves*, in which incoming traffic triggers outgoing traffic (which is not relayed), such as used by some forms of distributed denial-of-service tools [CE99]. These forms of stepping stones have different coincidence patterns than the interactive ones addressed by our algorithm, but a preliminary assessment indicates they may be amenable to detection on the basis of observing a local host that has long been idle suddenly becoming active outbound, just after it has accepted an inbound connection.

7 Acknowledgments

We would like to thank Stuart Staniford-Chen and Felix Wu for thought-provoking discussions, and in particular for the notion of deliberately introducing delay (§ 4.4); Ken Lindahl and Cliff Frost for their greatly appreciated help with gaining research access to UCB's traffic; and Mark Handley, Tara Whalen, and the anonymous reviewers for their feedback on the work and its presentation.

References

[Al94] S. Alexander, "Telnet Environment Option," RFC 1572, DDN Network Information Center, Jan. 1994.

[Bo90] D. Borman, "Telnet Linemode Option," RFC 1184, Network Information Center, SRI International, Menlo Park, CA, Oct. 1990.

[CE99] Computer Emergency Response Team, "Denial-of-Service Tools," CERT Advisory CA-99-17, Dec. 1999.

[DJCME92] P. Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin, "An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations," *Internetworking: Research and Experience*, 3(1), pp. 1-26, 1992.

[FJ94] S. Floyd, and V. Jacobson, "The Synchronization of Periodic Routing Messages," *IEEE/ACM Transactions on Networking*, 2(2), p. 122–136, April 1994.

[OR93] J. Oikarinen and D. Reed, "Internet Relay Chat Protocol," RFC 1459, Network Information Center, DDN Network Information Center, May 1993.

[PF95] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[Pa98] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Proc. USENIX Security Symposium*, Jan. 1998.

[PR83b] J. Postel and J. Reynolds, "Telnet Option Specifications," RFC 855, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PN98] T. Ptacek and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps, Jan. 1998.

[SH95] S. Staniford-Chen and L.T. Heberlein, "Holding Intruders Accountable on the Internet." Proc. IEEE Symposium on Security and Privacy, Oakland, CA, May 1995, pp. 39–49.

[YKSRL99] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen, "SSH Transport Layer Protocol," Internet Draft, draft-ietf-secsh-transport-06.txt, June 1999.

[Zi91] D. Zimmerman, "The Finger User Information Protocol," RFC 1288, Network Information Center, SRI International, Menlo Park, CA, Dec. 1991.