Detecting Hidden Causality in Network Connections

Jayanth Kumar Kannan
Jaeyeon Jung
Vern Paxson
Can Emre Koksal

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-30
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-30.html

December 19, 2005

Detecting Hidden Causality in Network Connections

Jayanthkumar Kannan, Jaeyeon Jung, Vern Paxson, Can Emre Koksal

December 20, 2005

Abstract

Upon success, certain network attacks manifest by causing the victim host to change its network-visible connection behavior, such as by starting a new service that the attacker probes to confirm success, "phoning home" to a host controlled by the attacker, or further propagating the attack (e.g., worms or spam relays). One characteristic of such change in network behavior is the presence of unusual causal relationships between connections. Based on this observation, we develop a statistical test that a network monitor can use to identify these causal relationships, and an accompanying set of filtering mechanisms to winnow down the full set of causal relationships to those that are unexpected. We evaluate our mechanism on two large Internet traces, finding that while its detection is incomplete (non-negligible false negatives), it unearths numerous instances of interesting activity. We also find that the rate of false alarms, while not low enough to enable automatic responses to intrusions, is only a few tens per day for a busy site that sees over 2.5 million connections a day.

1 Introduction

Upon success, certain network attacks manifest by causing the victim to change its network-visible connection behavior. One mechanism, popular with some "autorooter" toolkits (*e.g.,* [5, 6]), is used in exploits that cause the victim to begin accepting connections on a new port, which the attacker then probes to confirm success of the exploit. Another similar mechanism (*e.g.,* [4, 5]) is the use of "phone home" connections, where rather than starting a new service, the victim initiates an outbound connection to one of the attacker's nodes to register success of the exploit. For a different class of attacks, the victim instead propagates the attack in some fashion, *e.g.,* repeating the attack itself for network worms, relaying data for spam, conducting general probing for newly-compromised botnet zombies. In all of these cases, there is a *causal link* between the incoming network connection that compromises the host and subsequent incoming/outgoing network connections.

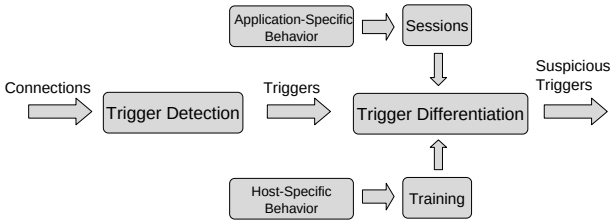


Figure 1: Overview of our Algorithm

Previous work on detecting such causality has primarily focused on end-host mechanisms for either detecting worms [7,9,15] or tracing the activity of intruders across the network [22]. While such mechanisms can be highly effective, they suffer from the deployment and management difficulties common to host-based techniques. In addition, the general problem with using causality to find attacks is that a causal link can transpire due to perfectly legitimate reasons, *e.g.,* a web proxy propagating a request it receives. In fact, as we will show, the overwhelming majority of causal relationships in real traffic arise due to such legitimate reasons. Schemes that impose the burden of distinguishing between legitimate and malicious causal links on the administrator may thus prove unmanageable.

In this work, we develop a mechanism for identifying *malicious* causal links that can be implemented at a monitoring point in the network *without* any modifications to the end-hosts. Our mechanism operates on a per-connection basis and aims to detect malicious *triggers*. We define a trigger as a pair of connections that are causally related; the first connection "triggers" the occurrence of the second connection. As shown in Figure 1, there are two main components in our mechanism to detect malicious triggers: a *trigger detection* mechanism that finds triggers in a light-weight manner and a *trigger differentiation* mechanism that distinguishes between legitimate and malicious (or at least unexpected)

triggers.

We base our trigger detection mechanism on modeling the temporal characteristics of connection arrivals. The main observation is that, in comparison with unrelated connections, causally related connections tend to occur "close" to each other. We identify such causal relationships by using a probabilistic test framed in terms of testing for independence between Poisson processes. An important point is that while it is known that the arrivals of individual network connections are *not* well-modeled as Poisson processes, connections can be suitably aggregated into *sessions*, whose arrivals often *are* well-modeled as Poisson [16, 18]. Thus, our approach requires identifying the session structure of network traffic.

We base our trigger differentiation technique on an analysis of legitimate triggers typically seen in real traffic. We identify two main causes of such triggers, application-specific behavior and host-specific behavior, and use the techniques of session-level parsing and host-level training to capture these behaviors. Our evaluation shows that these two techniques capture most legitimate activity; they generally only spare activity that an administrator would at least consider "interesting".

We note that an attacker cognizant of our detection algorithm can evade detection by suitably altering the timing of their triggered connections. We present our approach as a "tool in the toolbox" in the unending arms race between attackers and defenders, rather than as a bulletproof solution. Further, our trigger detection and differentiation mechanisms are decoupled, which means that the latter can be used with other means of detecting causality.

We evaluated our scheme using several weeks of traces collected at the border of two institutions, one with 272 hosts and 112,500 connections a day, the other with 7,879 hosts and 2,700,000 connections a day. Our algorithm identifies about 4 and 15 incidents per day in these datasets respectively. Although only a few of these incidents were possible intrusion attempts, several were of significant interest to administrators, *e.g.,* unauthorized relays and unknown peer-to-peer applications. Thus, the triggers identified by our mechanism do not necessarily imply malicious intent, but often signal unusual host activity that merits administrator investigation. These evaluation results indicate that our mechanisms may also be suitably tailored for detecting specific kinds of casual relationships, such as those that arise due to peer-to-peer applications and spam relaying. Such triggers, though not indicative of externally launched attacks, may signify violation of a site's security policy from within the site.

The outline for the rest of the paper is as follows. We begin with some background in Section 2 and discuss related work in Section 3. We propose and evaluate our trigger detection mechanism in Sections 4 and 5, do the same for our trigger differentiation mechanism in Sections 6 and 7, and discuss their implementation in Section 8. We cast our work in a wider perspective in Section 9 and conclude in Section 10.

2 Background

We now detail the problem setting by specifying the traffic characteristics available as input to our algorithm, and then present our terminology for describing triggers.

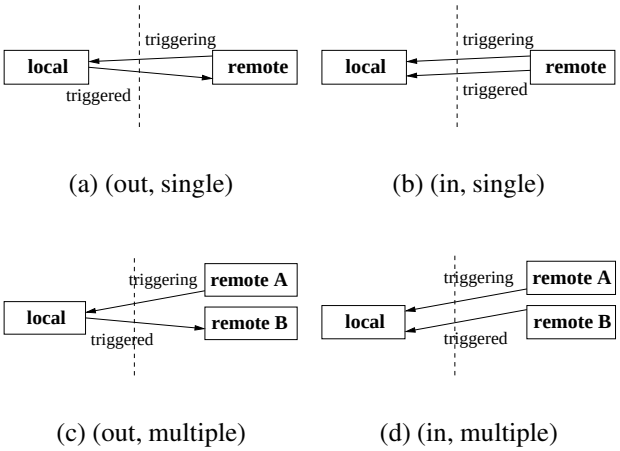
2.1 Input Traffic Characteristics

Our algorithm works using connection-level information. We assume we have a network monitoring vantage point that sees both the original attack traffic and the ensuing triggered traffic. For every TCP connection, this monitor reports the IP addresses of the local and remote hosts, direction (incoming/outgoing), timing information (start time, duration), and the connection's status (whether successfully established). We note that similar information could be generated for UDP flows as well, but have evaluated our mechanism only for TCP traffic.

We denote a connection C by the tuple (*dir, proto, remote-host, local-host, start-time, duration*). *dir* is "**in/out**" indicating whether the connection was initiated by the local (internal) host or the remote (external) host. *proto* specifies the destination port X of the connection. It is set to "priv-X" for ports $X < 1024$ which are usually only available to privileged users, and to "other-X" to indicate port numbers $X \geq 1024$. If X is a well-known port (*e.g.,* 22, 80) known to run a specific service, then for readability we set *proto* to the name of the service, such as **ftp** or **http**. *remote-host/local-host* is the IP address of the remote/local host, and *start-time* and *duration* denote the beginning times and duration, respectively. We define the *type* of a connection as the tuple (*dir, proto*). We allow some fields to be absent in our tuple notation for connections; the value of such omitted fields will be clear from context.

2.2 Trigger Terminology

We define a *trigger* as a pair of causally related connections (C_1, C_2), where C_1 is the "triggering" connec-



(a) (out, single) (b) (in, single)

(c) (out, multiple) (d) (in, multiple)

Figure 2: Taxonomy of triggers. The solid lines are connections, and the dotted line indicates our vantage point.

tion and C_2 is the "triggered" connection. For the triggers of interest to us, the "triggering" connection is always an incoming one, signifying an externally initiated connection attempt. Note that the concept of a trigger can be generalized to more than two causally related connections. This is useful, for example, when studying worm attacks, where the incoming connection C_1 triggers several outgoing probes $C_2, \ldots, C_n$. In defining a trigger, we stick to the simpler notion of pairwise causality, with the implicit understanding that a set of triggers, $(C1, C_2), \ldots, (C_1, C_n)$, which share the same triggering connection C_1, are related to one another. Later, we will introduce the concept of a *session* which consists of a set of causally related connections.

We classify triggers based on two parameters: the direction of the triggered connection and the number of remote hosts involved in the trigger (Figure 2).

- *(out, single)*: outgoing triggered connection and same remote host involved in both connections.
- *(in, single)*: incoming triggered connection and same remote host involved in both connections.
- *(out, multiple)*: outgoing triggered connection and multiple remote hosts involved in the connections.
- *(in, multiple)*: incoming triggered connection and multiple remote hosts involved in the connections.

We define the *type* of a trigger as the category to which it belongs in this taxonomy. For convenience, we also use the notation "$\Rightarrow$, $\Leftarrow$, $\Rightarrow\Rightarrow$, $\Leftarrow\Leftarrow$" for the type of a trigger, by using the symbols "$\Leftarrow, \Rightarrow$" to encode the direction of the triggered connection, and repeating the symbol twice if more than one host is involved. We define the *protocol-type* of a trigger as its type followed by the protocols involved in the connections of the trigger. As an example, a trigger $(C1, C2)$ involving the triggering connection $C_1 = (\text{in}, \text{http}, H_1)$ and the triggered connection $C_2 = (\text{out}, \text{smtp}, H_2)$ would have a type *(out,multiple)*, denoted by "$\Rightarrow\Rightarrow$", and the protocol type "$\Rightarrow\Rightarrow$ http smtp".

The motivation behind this classification is that each category of triggers has different properties, in terms of their frequency in legitimate traffic, their frequency in malicious traffic, and, most importantly, the difficulty of detection. In the above list, these categories are ordered roughly by the difficulty of detection. An outgoing triggered connection is easier to detect compared to an incoming triggered connection, since it is less common that a host exhibits both server and client behavior than purely server behavior. Triggers involving two different remote hosts are more difficult to detect since the number of pairs of candidate connections is generally larger.

3 Related Work

Of the three broad approaches to detecting malicious activity—misuse detection (*e.g.,* [19]), specification-based detection (*e.g.,* [23]), and anomaly detection (*e.g.,* [10])—our work is a form of the third.

The literature includes a number of statistical techniques for detecting network attacks that occur either in high volume or involve repeated activity. Barford *et al.* [3] use wavelet analysis to detect large-scale network events such as flash crowds, denial of service (DoS) attacks, network outages, and Jung *et al.* [11] use statistical analysis to detect flash crowds based on characteristics observed at Web servers, such as client source addresses, traffic volume, and file references. Krishnamurthy *et al.* [13] develop sketch-based change-point detection to identify shifts in traffic pattern indicative of network anomalies. Jung *et al.* [12] present a scheme based on Sequential Hypothesis Testing for identifying port scans based on the observation that port scan connection attempts have a lower probability of success than legitimate connections. Xie *et al.* [25] aim to find the origin of a worm attack by performing a random walk on the graph of infector-infectee relationships.

In comparison with these works, where the same kind of deviant behavior occurs repeatedly or en masse, the kind of malicious behavior we intend to detect may occur very rarely. We can subdivide the literature on identifying such less conspicuous attacks into those requiring end-host modifications versus those that operate purely in the network.

Minos [9], TaintCheck [15], Vigilante [7], and StackGuard [8] all prevent/detect attacks using end-host instrumentation. GrIDS [22] uses software running at the end-host to trace the activity of an intruder across a network, particularly useful for post-mortem forensic analysis. Warrender *et al.* [24] perform statistical analysis of system calls observed at a host in order to identify intrusions. NIDES [1] uses various process characteristics, such as CPU time, file accesses, measured at the end-host, to detect attacks. Compared to our work, these techniques can detect a broader class of attacks, but with the limitation of requiring per-host installation and management. Also, our work only detects attacks that succeed, since network characteristics can change only upon a successful attack.

Several other works identify low-volume attack activity using only analysis of network characteristics. Mahoney *et al.* [14] build models of IP and TCP header fields, as well as application-level fields for protocols like HTTP. Their approach uses Bayesian estimation techniques for detecting deviations from this model. ADAM [2] also performs Bayesian analysis of network layer and transport layer fields in order to detect attacks. Zhang *et al.* [26] detect stepping stones, a specific kind of malicious activity wherein attackers use a previously compromised machine to obscure their identity while attacking other machines. Their approach uses only the packet timing structure of interactive connections to find concurrent connections that are operating in synchronization.

Compared to these works, ours is most similar in spirit to stepping stone detection [26]. We focus on detecting causal relationships in network traffic, using a statistical framework that relies on little more than properties of the Poisson arrival process. Compared to the Bayesian mechanisms used in [2, 14], we can establish a bounded false positive rate in terms of how often our framework reports that a causal relationship exists between two unrelated connections.

4 Trigger Detection

Our trigger detection algorithm is based on the observation that if two connections are causally related, then the arrival of the triggered connection is likely to be "unusually" close to the arrival of the triggering connection. We now describe a more formal statement of this intuition.

4.1 Modeling Normal Traffic

The problem of distinguishing between triggered and normal connections can be phrased in terms of hypothesis testing [20]. In this formulation, the arrival time of a connection is used to choose between the *null* hypothesis that this connection is normal, and the *alternative* hypothesis that this connection is triggered. Modeling the alternative hypothesis, however, requires capturing the statistical dependence of the arrival time of the triggered connection on the arrival time of the triggering connection, but this may depend on the semantics of the trigger itself. Fortunately, as we will show shortly, the null hypothesis is easier to model, therefore our inference algorithm finds triggers by building a model for the arrival characteristics of untriggered (normal) connections. We can then identify connections whose arrivals deviate from this model as triggered connections.

At the heart of our approach is the empirical observation that the arrival of *user-initiated sessions* of network activity is well-modeled as a Poisson process, stationary over time scales of an hour [16, 18]. The term *session* here means a collection of aggregate network activity (such as the individual HTTP connections that make up a user's Web surfing), and *user-initiated* means sessions that are instigated by human activity rather than machine activity (such as periodic daemons). Guided by these findings, we maintain a model for connection arrivals by estimating rates for different types of connections over a sliding window of duration $\mathsf{T}_{rate} = 1$ hr.

Our model views the activity of local hosts as independent, and, further, is *type-based* in that we maintain independent notions of connection rates for the different types of applications in which each host participates. Our definition of connection type includes not only the application type but also the connection direction, which accounts for the difference in the arrival characteristics of clients and servers.

4.2 Detection Algorithm

We now describe a statistical test that identifies triggers by using our model of normal traffic.

Consider the arrival of two connections C_1, C_2 whose types are denoted by T_1, T_2. Denote the arrival rates of these connection types as λ_1 and λ_2. Let the interarrival time between C_1 and C_2 be x. Now suppose that C_1 triggers C_2. In this case, we expect that x is significantly lower than the case when C_1 and C_2 are unrelated. Our strategy is therefore to estimate the probability P of ob-

serving an interarrival x for the null hypothesis of the connections being unrelated, and to classify the pair as a trigger if P is less than a confidence threshold α.

Consider the unusual event that two connections of type T_1, T_2, whose arrivals follow *independent* Poisson processes with rates λ_1, λ_2, arrive within a duration x of one another. We denote by $P[T_1, T_2, x]$, the probability that such an event occurs at least once in one unit of time. Given such a formulation, trigger detection proceeds as follows. We first categorize connections into different types; estimate rates for each of these types, and then use these computed rates along with the threshold α to detect triggers. More specifically, on the arrival of a connection C of type T at a local host L, the following actions are performed:

- Let the connections observed at L in the previous $\mathsf{T}_{trigger}$ seconds be $C_1, C_2, \ldots, C_n$, with types $T_1, T_2, \ldots, T_n$. $\mathsf{T}_{trigger}$ is a user-defined threshold on how far apart in time the two connections of a trigger can be.
- Estimate the rate of connection arrivals at L for every connection type within the past T_{rate} seconds. This rate is estimated simply as the average interarrival time between connections of a specific type, over the interval of duration T_{rate}.
- For $1 \leq i \leq n$, compute $P[T_i, T, x_i]$, for x_i the interval between the arrival of C_i and C.
- If $P[T_i, T, x_i] < \alpha$, classify (C_i, C) as a trigger.

The key step in the above algorithm is the use of the parameter α as a probability threshold. Too low a value for this parameter may mean that we may miss certain triggers (false negatives) and too high a value may lead to identification of spurious triggers (false positives). The false negative rate is difficult to characterize analytically due to the lack of a statistical model for the arrival of triggered connections, so we rely on empirical analysis for evaluating it. Our choice for $P[T_1, T_2, x]$ however allows us to provide a bound on false positives, as developed in the next section.

4.3 False Positives

The following theorem establishes a bound on the number of false positives in our detection mechanism.

Theorem 1. *$P[T_1, T_2, x]$ can be written in terms of the connection arrival rates λ_1, λ_2, as $\lambda_1 \lambda_2 x$. With this setting, the number of false positives at a given host over any time-window of duration T is bounded from above by $\frac{M(M+1)\alpha\mathsf{T}}{2}$, where M is the number of different types of connections seen at that host.*

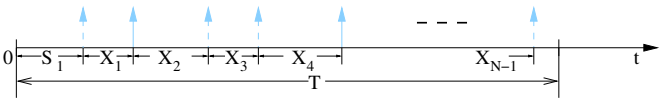


Figure 3: Poisson process with N arrivals over time T.

We prove this claim in three steps. First, we consider the case of a single Poisson process with rate λ, and compute the probability that it has two events that fall less than time x apart within a time interval T. Second, we adapt this lemma to the case when there are two independent Poisson processes of rates λ_1, λ_2. Finally, we extend these results to the case when there are M independent Poisson processes.

Lemma 1. *Let $\{X_i, i \leq N(\mathsf{T})\}$ be the interarrival times of a Poisson process with an arrival rate λ observed in the interval $(0, \mathsf{T}]$ where there are $N(\mathsf{T})$ arrivals. Then, $\Pr[X_i < x$ for some $i \leq N(\mathsf{T})] \leq \lambda^2 \mathsf{T} x$.*

Proof. Define X_i as the i^{th} interarrival time of this arrival process (shown in Figure 3). We first evaluate the probability that a pair of arrivals occur within x seconds of each other in this window of T seconds. First, we evaluate the conditional probability given $N(\mathsf{T}) = N$. Let S_1 be the time of the first arrival. We have:

$$\Pr[S_1 > x | N(\mathsf{T}) = N] = \left(\frac{\mathsf{T} - x}{\mathsf{T}}\right)^N = \left(1 - \frac{x}{\mathsf{T}}\right)^N$$

This holds since the distribution for arrival epochs follows the order statistics for a joint uniform distribution of N random variables [20]. Therefore:

$$\begin{aligned} \Pr[S_1 < x | N(\mathsf{T}) = N] &= 1 - \left(1 - \frac{x}{\mathsf{T}}\right)^N \quad (1) \\ &\leq 1 - \left(1 - N\frac{x}{\mathsf{T}}\right) \quad (2) \\ &= \frac{Nx}{\mathsf{T}} \end{aligned}$$

where (2) follows from (1) by ignoring the following higher order terms:

$$\binom{N}{2}\left(\frac{x}{\mathsf{T}}\right)^2 - \binom{N}{3}\left(\frac{x}{\mathsf{T}}\right)^3 + \cdots \geq 0$$

Since the time for the first arrival is identically distributed with the i^{th} interarrival time,

$$\Pr[X_i < x | N(\mathsf{T}) = N] \leq N\frac{x}{\mathsf{T}}, \text{ for } 1 \leq i \leq N - 1$$

Next, let us look at the probability that there exists such an interarrival within N arrivals.

$$\begin{aligned}
&\Pr[X_i < x \text{ for some } i \le N(\mathsf{T}) | N(\mathsf{T}) = N] \\
&\quad = \Pr[\ \cup_i (X_i < x | N(\mathsf{T}) = N)\ ] \\
&\quad \le \sum_i \Pr[X_i < x | N(\mathsf{T}) = N] \qquad (3) \\
&\quad = (N-1)N\frac{x}{\mathsf{T}}
\end{aligned}$$

where (3) follows from the union bound [20] (the probability of the union of a set of events is at most the sum of the probability of the individual events).

Lastly, we find a bound for the *unconditional* probability that there exists an interarrival of size smaller than x in the entire window of T seconds.

$$\begin{aligned}
&\Pr[X_i < x \text{ for some } i \le N(\mathsf{T})] \\
&\quad \le \frac{x}{\mathsf{T}} \mathrm{E}[(N(\mathsf{T})-1)N(\mathsf{T})] \\
&\quad = \frac{x}{\mathsf{T}} \left[\mathrm{E}[(N(\mathsf{T}))^2] - \mathrm{E}[N(\mathsf{T})]\right] \\
&\quad = \frac{x}{\mathsf{T}} \left[\mathrm{Var}[N(\mathsf{T})] + (\mathrm{E}[N(\mathsf{T})])^2 - \mathrm{E}[N(\mathsf{T})]\right] \\
&\quad = \frac{x}{\mathsf{T}} (\mathrm{E}[N(\mathsf{T})])^2 = \lambda^2 \mathsf{T} x \qquad (4)
\end{aligned}$$

where the last two equations rely on the properties of the Poisson random variable: the variance is equal to the mean, which is $\lambda\mathsf{T}$. Note that the bound we find becomes tight if the average interarrival time is large compared to x, i.e., $\lambda x \ll 1$. In this regime, the probability that two or more occurrences of such interarrivals is unlikely. Thus, the higher order terms in the binomial expansion are negligible and the union bound is tight. □

We now generalize the previous lemma to the case when there are two types of connections, type T_1 and type T_2, seen at a given local host.

Lemma 2. *Given two independent Poisson processes P_1 and P_2 with rates λ_1 and λ_2, observed over time T, the probability that an arrival in P_1 occurs within a duration x of an arrival in P_2 is bounded by $\lambda_1\lambda_2\mathsf{T}x$.*

Proof. Consider the aggregate process consisting of arrivals from both P_1 and P_2. Since this aggregate process is Poisson with rate $(\lambda_1 + \lambda_2)$, Lemma 1 bounds the probability that an interarrival of less than x occurs as:

$$\Pr[\text{interarrival} < x] \quad \le \quad (\lambda_1 + \lambda_2)^2 \mathsf{T} x$$

Since we also require that the two connections belong to two different types, we can evaluate the probability, P_{un}, that in a window of T seconds, we observe the unusual event that a type T_1 arrival is followed by a type T_2 arrival within x seconds:

$$\begin{aligned}
P_{\text{un}} &\le (\lambda_1 + \lambda_2)^2 \mathsf{T} x \cdot \frac{\lambda_1}{\lambda_1 + \lambda_2} \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \\
&= \lambda_1 \lambda_2 \mathsf{T} x. \qquad (5)
\end{aligned}$$

This formula uses the fact [20] that the probability of a given arrival in this aggregated process being of type T_1 and type T_2 is $\frac{\lambda_1}{\lambda_1+\lambda_2}$ and $\frac{\lambda_2}{\lambda_1+\lambda_2}$ respectively. Note that, due to the properties of the aggregated Poisson process, these probabilities are independent of the connection arrival times (which we have already conditioned on), and are also independent of each other. □

As we have learned from experience not to always trust these sorts of somewhat subtle derivations, we also validated Lemma 2 (and by implication Lemma 1) using Monte Carlo simulations; see Appendix A.

Theorem 1 follows directly from Lemma 2. To see this, consider the arrivals of two specific types T_1, T_2 of normal connections within a time-window of T. Over this duration T, the probability that at least one type T_1 event occurs within duration x of a type T_2 event (or, vice-versa) is upper-bounded by $\lambda_1\lambda_2 x\mathsf{T}$. This implies that the probability of such an unusual event per unit time is upper-bounded by $\lambda_1\lambda_2 x$, which is exactly the formula for $P[T_1, T_2, x]$. Note that due to the binomial approximation used in Lemma 2, this relation holds exactly if $(\lambda_1+\lambda_2)x \ll 1$, and is otherwise a upper-bound. Further note that $P[T_1, T_2, x]$ is a likelihood measure and need not sum to 1.

To prove the second part of the theorem, note that we report a connection pair only if $P[T_1, T_2, x] < \alpha$, which is equivalent to the condition $x < \alpha/(\lambda_1\lambda_2)$. Plugging this into Lemma 2, one can see that α is an upper-bound on the number of false positives per unit time. A simple union bound argument shows that if there are M different types of connections, the number of false positives over time T is bounded by $\frac{M(M+1)\alpha\mathsf{T}}{2}$. In practice, we choose the unit of time for measuring α as an hour, and have found that $\alpha = 0.01$ per hour works well. The number of false positives is typically lower than the worst-case bound proved above.

Although the derivation in Lemma 2 assumes stationarity over the duration T, our results apply even otherwise. The interval T can be divided into sub-intervals where the stationary assumption holds, and a union bound argument across these sub-intervals yields the re-

Parameter	Description	Setting	How Determined
T_{rate}	Duration for rate estimation	3,600 sec	[16, 18]
$\mathsf{T}_{trigger}$	Maximum interval between the sessions in a trigger	500 sec	Empirical
α	False positive threshold	0.01	Calibration

Table 1: Parameters in Trigger Detection

quired result. The key point is that our false positive rate is a constant α and is *independent* of the varying connection arrival rates. As long as we use *instantaneous* arrival rate estimates in the probability test, the *instantaneous* false positive rate is bounded by α.

5 Evaluating Trigger Detection

In this section, we evaluate the effectiveness of the trigger detection mechanism. The emphasis is on assessing its performance in terms of false negatives, as the assessment of false positives is done later when we add the trigger differentiation mechanism, whose role is to reduce false positives and weed out uninteresting but correct detections.

Datasets: We experimented with two datasets, D_1 and D_2, collected at the borders of two sites, S_1 and S_2, that have, in the datasets, 272 and 7,879 active local hosts, respectively. Both traces are address-anonymized versions of connection logs gathered by the Bro intrusion detection system [17]. Trace D_1 spans 2 months, corresponding to traffic seen at S_1 in January and February 2005, at an average of 112,500 connections per day. Trace D_2 spans one month, February 2005, with an average of 2,700,000 connections per day. Both sites are monitored by a security team. S_1's security team reports a very low rate of successful attacks (a few a year), attributed to the site's fairly homogeneous system administration and limited set of services. S_2 experiences an average of one successful attack (possibly involving multiple local hosts) per week or so. Both sites use firewalls that block certain incoming services, but have a general "default allow" posture similar to educational institutions, and both have users who run services on their machines. Apart from the difference in scale, S_1 and S_2 also qualitatively differ in that the latter is a more diverse environment with many applications, some of which would not be typically seen in other environments.

Since our algorithm involves tunable parameters, in general we used the first 2 weeks of data from D_1 and D_2 as *calibration data* for setting these parameters. We then used the remaining data as *validation data* for assessing the performance of the calibrated algorithm.

Calibration: Table 1 lists the parameters used by our detection mechanism. We fixed T_{rate} using the finding reported in [16, 18] that arrivals of user sessions generally follow stationary Poisson processes over time scales of an hour. In our implementation, in order to give adequate time for the rate estimates used in our statistical tests to converge, we apply the tests only after a stabilization period of T_{rate} seconds has elapsed since starting. Most successful attacks for which we have data involve a triggered connection within 100 sec of the triggering connection, so we conservatively set $\mathsf{T}_{trigger}$ to 500 sec.

For calibrating α, we chose different types of known legitimate triggers and set α so as to obtain low false negative performance in finding these triggers in the calibration data. We focus primarily on low-volume, unusual triggers for calibration, since such triggers resemble malicious triggers in frequency of occurrence. For D_1, however, use of such services was not readily apparent, so to calibrate it we chose the trigger of protocol-type "$\Rightarrow$ smtp ident". This occurs at mail servers that, upon receiving an *smtp* connection, establish an *ident* connection back to the remote host, usually in order to validate the originating domain name.

For D_2, we selected three hosts running non-standard services, which lead to three unusual triggers, "$\Rightarrow\Rightarrow$ smtp razoragent , "$\Rightarrow\Rightarrow$ other-8080 http" , and "$\Leftarrow$ http other-9303". The first is seen at a mail server that forwards received mail to a spam filter called RazorAgent running on a different machine. The second occurs at a web proxy set up by a user, and the third due to a web interface to a service running on the host on port 9303.

Finally, for both D_1 and D_2 we also used triggers (C_1, C_2) with $C_1 = (in/out, ftp, H)$ and $C_2 = (in/out, ftp - data, H)$ for calibration. These occur due to the FTP protocol where the FTP user-session connection triggers several secondary data-transfer connections in either direction. This is a particularly challenging trigger to detect since FTP sessions can last for a long time (hundreds of seconds). Our statistical test has

Trigger Type	SMTP (S_1)			FTP (S_1)		
# Hosts (Conn.)	1 (308,818)			34 (44,468)		
Setting of α	0.001	0.01	0.1	0.001	0.01	0.1
False Neg. (# Hosts)	**1**	**1**	0	11	8	6
False Pos. (# Conn)	1	1	110	40	129	385

Table 2: Calibration Results at Site S_1 (using first 2 weeks of D_1)

Trigger Type	RazorAgent (S_2)			WebProxy (S_2)			Service-9303 (S_2)			FTP (S_2)		
# Hosts (Conn.)	1 (196,033)			1 (1,915)			1 (912,137)			591 (15,526)		
Setting of α	0.001	0.01	0.1	0.001	0.01	0.1	0.001	0.01	0.1	0.001	0.01	0.1
False Neg. (# Hosts)	**1**	0	0	0	0	0	0	0	**1**	372	273	157
False Pos. (# Conn)	8	25	66	6	9	10	937	2,440	5,929	38	217	865

Table 3: Calibration Results at Site S_2 (using 5 million connections in first 2 weeks of D_2)

a lower chance of success in finding such triggers with long interarrival times. The FTP application is therefore a stress test for our algorithm; we expect that most malicious triggers will involve much shorter interarrivals.

Calibration Results: Tables 2 and 3 present our calibration results for three values of $\alpha = 0.1, 0.01, 0.001$. For S_2, since the volume of reported triggers is high, we only report on the first 5 million connections in D_2. We proceeded as follows. For each application, we first used the session-parsing approach developed in Section 6 to find all triggers related to the particular application. The tables list the number of local hosts involved in these triggers and the total number of connections over all of those hosts. We then measured the number of false negatives by counting the number of hosts for which our algorithm failed to find the *first* application trigger. We focus on the first trigger because we aim to detect successful attacks that may occur just once at a given host.

We see mixed results for false negatives. Since only one host provides the service for many of the triggers we tested, and because we only test for detection of the first instance of a trigger, for all services other than FTP the algorithm had exactly one chance to detect the activity. For these services, our algorithm succeeded for those entries where we list 0 hosts for false negatives, and failed where we list 1 host (in bold). We found the prevalence of failures somewhat discouraging, and investigated them further by testing for detections beyond just the first instance. This only improves the situation somewhat, with the false negative ratio of SMTP still at about $15 - 20\%$. More importantly, doing so gave insight into the underlying difficulty: most of the services we evaluated have quite high rates of connections. For these, the triggered secondary connection must arrive quite closely to be viewed as anomalous. (As an extreme example, consider a service that averages a thousand connections per second. Then any triggered connection must occur scant microseconds after one of these, or else it will be indistinguishable from a chance arrival.)

This is related to the fact mentioned earlier: our false positive rate is independent of the rates of the connections involved. This property is very desirable since this means that the false positive rate would be the same across hosts with widely varying rates. The trade-off, however, is that as the connection arrival rate increases, the interarrival duration for trigger detection is lower. In this light, the false negative rates are more understandable; a better approach would be to assess false negative rates for less popular services, but these are difficult to find in an unbiased fashion (i.e., other than by using our algorithm to find them in the first place). On the other hand, we note that our assessment in Section 7 will show that our algorithms *do* find a number of interesting triggers. Thus, even though we know from the analysis here that a non-negligible number of triggers will be missed, the detector clearly retains significant utility.

To evaluate false positives, we run our algorithm over all connections associated with the application of interest to see which triggers it reports. We count those not directly related to the application's semantics (which we determine using manual assessment) as false positives. Recall that our main interest here is in assessing false negatives rather than false positives. The false positive rates of final interest are those that appear after applying our trigger differentiation algorithm, developed and assessed later in the paper. That said, the false positive performance, as shown in Tables 2 and 3, is generally at most a few percent. The worst cases

are for Service-9303 (S_2) and FTP (S_1, S_2). In the former case, more than 95% of these false positives were due to triggers of the form "$\Leftarrow\Leftarrow$ http http", which involved connections from two different clients closeby in time. In some cases, it is unclear why this occurs (*e.g.,* in one instance, the connection pair was less than 200 ms apart, out of 4 connections in an entire hour). In the case of FTP, more than 80% of the false positives were triggers of the kind C_1 = *(in/out,ftp-data,H),* C_2 = *(in/out,ftp-data,H)*, that is, concurrent FTP data transfers, for which the *ftp* connection occurred too far in the past to be correctly associated with its data transfer. Other false positives are in fact not false positives but of the form $(in/out, ftp, H)(in/out, other\text{-P}, H')$, occurring due to multi-homing causing the data transfer to go to a different IP address.

Based on these results, we choose $\alpha = 0.01$ in order to choose an acceptable false negative and false positive performance. Most of the false positives under this setting are manifestations of various kinds of legitimate triggers, and our design of the trigger differentiation mechanism, developed in the next section, handles such legitimate triggers.

Validation Results: Running the parameterized algorithm against the validation datasets finds a very large number of triggers, as detailed in Appendix B. Manual inspection of a number of these indicates that most are indeed legitimate. In the interest of space, we do not analyze these in detail here; the main takeaways from the evaluation are that *(i)* network traffic is replete with triggered connections, and *(ii)* the great majority of these causal links occur due to perfectly legitimate reasons. Therefore, we need a trigger differentiation mechanism to deal with them, to which we now turn.

6 Trigger Differentiation

Until now we focused on detecting causality that likely exists between a pair of network connections. We now turn to the question of whether a given instance of such causality is actually of *interest* in terms of reflecting malicious activity or at least behavior out of the ordinary.

Our differentiation mechanism evolved from our experience running the trigger detection algorithm reported in Section 5. In a manner analogous to developing the trigger-detection mechanism, our strategy here is to capture the characteristics of legitimate triggers in a succinct fashion, and then report triggers that deviate from these characteristics as suspect. We classify legitimate triggers into two kinds, based on the typical behavior that leads to them:

- **Application-Specific Behavior**: Numerous applications initiate multiple connections per invocation, which leads to a sequence of causally related connections.
- **Host-Specific Behavior:** The notion of whether a given trigger is suspect has a critical host-specific element, due to differences in how hosts are configured

The first of these concerns capturing known *application session structure*. The second concerns deciding whether a form of *unknown* session structure is likely legitimate based on the host's past behavior. We discuss each in turn.

6.1 Application-Specific Behavior

Many application protocols consist of collections of connections that together comprise *sessions*. For example, common applications like FTP or HTTP often involve several TCP connections in response to a single user request (*e.g.,* a single invocation of an FTP client leads to a FTP-user connection followed by possibly several FTP-data connections; a single user Web browser session may lead to many HTTP connections). We expect network traffic to be replete with such sessions, and thus need to recognize their presence when possible, since the triggers due to such structures, while still reflecting causal links between connections, are uninteresting for the purpose of detecting attacks.

In addition, we *need* to identify session structure because the empirical basis for using Poisson processes to model network arrivals concerns the arrival of *sessions*, not connections [18]. Luckily, we can bootstrap our detection mechanism to be self-correcting in this regard: if we use it on non-Poisson arrivals, such as those from the individual connections of a not-yet-identified session, the detector will generally flag the arrivals as violating Poisson independence. This then allows us to inspect the violation and recognize the corresponding session structure. Once we have codified the structure (see below), we can then associate the individual connections together and treat their arrivals not as individual arrivals but as reflecting a single session arrival. These session arrivals then will then give us Poisson behavior.

In this light, we define the notion of a *session* as an ordered sequence of connections $(C_1, C_2, \ldots, C_m)$ that are causally related to each other. Ideally, each session captures all the network activity due to a single user-level action. We define the type and arrival time of a

session S as the type and arrival time of the first connection in S.

We now describe a characterization of legitimate sessions, and then discuss how to incorporate this characterization into our trigger detection mechanism.

6.1.1 Characterizing Application Behavior

We have found that the structure of legitimate sessions, in terms of the types of connections that constitute them, is generally quite simple, so that we can capture the gist of it using regular expressions.[1] A regular expression (regexp) for a particular application defines the set of sequences of connections that can arise as a result of legitimate behavior of that application. For example, we can specify the sessions associated with FTP with the following regexp:

*(in, ftp) (out, ident)? (in/out, ftp-data)**

Such regexps are written using the type of the individual connections, expressed in tuple form, as the alphabet (though see below). This one captures an optional *ident* connection made by some FTP servers as well as the multiple *ftp-data* connections in a single *ftp* session. These *ftp-data* connections may occur in either direction, corresponding to FTP's active and passive data transfer modes.

We can extend this notation to support the use of dynamic port numbers in the regexp definition. For example, we use the following regexp for certain RPC-based services: *(in, portmap) (out, ident)? (in, P) (in, P+1).* This regexp illustrates the case where a client contacts the *portmap* service in order to find the port number P where a desired service is listening. The protocol for this service involves contacting ports P and $P + 1$.

Another extension we need is to introduce the notion of the hosts involved in the connections. Some sessions always involve the same remote host, whereas others can involve multiple hosts. As an example of the latter, we use the regexp *(out,aol,*H_1*) (out,http,*H_2*)* for the AOL instant messenger protocol, where the AOL client is typically redirected to a specific website upon connecting to the AOL chat server. (Here, the service *aol* refers to 5190/tcp.) This regexp specifies that the *aol* and *http* connections need not involve the same host.

We also found that, for some applications, a single user session leads to multiple invocations of the application-level protocol. This holds, for example, for HTTP, RTSP and SMTP. For this reason, we collapse nearby sessions of the same type involving the same remote host into a single *aggregated* session.

[1]Per Section 8.2, we actually use extended, rather than pure, regular expressions.

6.1.2 Session-Level Analysis

The naive way of using our definition of sessions is as a filter over the results of our connection-level trigger detection mechanism; we only flag triggers as suspect if they do not match a known, legitimate session definition. As noted above, since network traffic contains significant session structure, the more powerful use of the notion of sessions is to switch our trigger detection to analyzing session arrivals rather than connection arrivals. Given a sequence of connections $C_1, C_2, \ldots$, we use the regexps to parse it into a sequence of sessions $S_1, S_2, \ldots$, where the type of each session is set to the type of its first connection. We can then directly apply the algorithm discussed earlier in Section 4 to this sequence. The algorithm maintains rate estimates for each type of session, uses the duration between session arrivals rather than connection arrivals in the probability computation, and reports sessions that are unusually close to one another as suspicious. The analytical bound on the number of false positives in Section 4.3 remains applicable to session-level detection as well.

In our implementation, we use two timeouts to prevent sessions from becoming unduly extended (after all, regular expressions match infinitely long strings), and to coalesce nearby sessions. Any two consecutive connections C_i, C_{i+1} in a session must satisfy the property that C_{i+1} arrives no later than $\mathsf{T}_{session}$ seconds after the termination of C_i. Further, we only coalesce sessions if they are less than T_{aggreg} seconds apart. Our settings for these parameters are guided by [16] which studied aggregation for HTTP connections, and our empirical results; see Table 4.

6.2 Host-Specific Behavior

Our analysis of real-life datasets made it amply clear that the notion of legitimacy does not have a tidy "one size fits all" property. A trigger of the form "$\Rightarrow\Rightarrow$ smtp smtp" is a very mundane occurrence at an SMTP relay server, but would be a very suspicious happening at a user's desktop. For this reason, we must maintain additional per-host state that indicates what kind of triggers constitute expected behavior. This state is essentially a white-list generated by observing past behavior over a fixed training period of N days. When our mechanism is used over several days of traffic, no

Parameter	Description	Setting	How Determined
$\mathsf{T}_{session}$	Maximum interval between two connections in a session	200 s	Empirical
T_{aggreg}	Maximum interval between sessions in an aggregated session	100 s	[16]
N	Duration of Training Period	7 days	Empirical

Table 4: Parameters in Trigger Differentiation

triggers are reported to the user during the first N days, and only the white-lists are populated. After the training period, triggers that do not match these white-lists will be reported. We use three types of heuristics for populating white-lists that prove useful in practice:

- **Trusted trigger types:** Extending our definition of a protocol-type, the protocol type of a trigger $T = (S_1, S_2)$ is the string "$N\ T_1\ T_2$", where T_1, T_2 are the types of sessions S_1, S_2, and N is $\Rightarrow, \Leftarrow$ etc depending on the type of the trigger. For a newly detected trigger, we refrain from flagging it if the local host saw another trigger of that protocol-type in the past N days. This white-list is very useful for identifying relays for various applications, such as SMTP, HTTP. For such hosts, we white-list triggers of the protocol-type "$\Rightarrow\Rightarrow\ X\ X$" where $X = \texttt{http/smtp}$.
- **Trusted remotes:** We assume that any remote host to which a local host L *initiated* an untriggered connection in the past N days is trusted by L. We assume that such a remote host is not likely to source attacks against that local host L, and incorporate it in the white-list for L. In a production environment, we find that collaborative efforts can lead to unusual interactions, such as backups or security auditing between local and remote hosts. Automatically white-listing such hosts helps avoid flagging these triggers, and saves the labor of manually adding regexps for these unusual applications. In our datasets, we encountered several specialized and uncommon applications used for such purposes, on whose protocols we could find little documentation.
- **Trusted ports:** Any port on which a local host has previously accepted an untriggered connection is assumed to reflect a legitimate service. Therefore, we treat *(in, single)* and *(in, multiple)* triggers for which the second (triggered) connection corresponds to such a port as uninteresting. This prunes out triggers that may occur at hosts running multiple services in which two different services are accessed during a short time interval.

Note that we use only untriggered connections in setting up the white-lists of ports and remote hosts. This protects against a simple way to game our system; otherwise, a malicious host can incorporate itself into the white-list.

7 Evaluating Trigger Differentiation

We first describe how the parameters in our trigger differentiation mechanism are set, and then the calibration and validation results using our datasets.

Parameters: Table 4 lists the parameters used by our algorithm. We set $\mathsf{T}_{session}$ by evaluation against our datasets. We found that some FTP and login sessions can involve long durations between consecutive connections, which calls for using a high $\mathsf{T}_{session}$ (in our experience, the algorithm is not very sensitive to this parameter). We set T_{aggreg} based on the study of aggregate HTTP connections in [16]. The parameter N corresponds to the training period used to capture past behavior. We used $N = 7$ days to capture both weekly patterns as well as diurnal fluctuations, which from earlier experimentation we had found to be potentially significant.

Our implementation incorporates 29 regular expressions corresponding to 29 different applications. We identified these by a manual analysis of triggers oft-reported by our detection algorithm. 12 of these suffice for S_1, but the others were needed for the more diverse site S_2. See Appendix D for a list of these.

Calibration: Our choice to maintain three kinds of white-lists—trusted trigger types, trusted remotes, and trusted ports—was motivated by our calibration results, shown in Table 5. The columns list the results of session-level analysis along with none/one/all of the three kinds of white-lists. Note that since our training period is 7 days, we used the first week of the calibration trace to generate the white-lists, and evaluated the results using the second week.

Observe that the white-list of trusted remotes is most useful in dealing with *(out,single)* triggers, while the trusted-ports white-list works best for *(in,single)* and *(in,multiple)* triggers. This is to be expected from our design; the former handles unusual applications, while the latter captures accesses to multiple services running at the same host within a short period of time. The trusted-

$\mathbf{S}_1$	Sessions	Sessions	Sessions	Sessions	Sessions
White-list Used	*None*	*Trusted Trigger Types*	*Trusted Remotes*	*Trusted Ports*	*All*
All Triggers	388	66	330	60	12
(out,single)	4	4	1	4	1
(in,single)	24	10	22	3	2
(out,multiple)	53	19	10	53	9
(in,multiple)	307	33	297	0	0
$\mathbf{S}_2$	Sessions	Sessions	Sessions	Sessions	Sessions
All Triggers	19,344	801	16,527	1,306	219
(out,single)	39	23	7	39	7
(in,single)	4,069	128	2,392	67	34
(out,multiple)	1,129	313	225	1,129	170
(in,multiple)	14,107	337	13,903	12	8

Table 5: Calibration Results at Sites S_1 and S_2 (over second week of D_1, D_2)

	(out,single)	(in,single)	(out,multiple)	(in,multiple)
Number	4	426	116	5
Top 5 (histogram by protocol-type of trigger) for **Site** $\mathbf{S}_1$	1 $\Rightarrow$ other-4525 otherdyn 1 $\Rightarrow$ other-4212 otherdyn 1 $\Rightarrow$ other-3689 otherdyn 1 $\Rightarrow$ other-1572 otherdyn	180 $\Leftarrow$ other-2200 otherdyn 137 $\Leftarrow$ other-32774 otherdyn 33 $\Leftarrow$ other-2200 other-898 25 $\Leftarrow$ other-32774 other-898 7 $\Leftarrow$ other-1025 otherdyn	35 $\Rightarrow\Rightarrow$ other-36352 otherdyn 17 $\Rightarrow\Rightarrow$ https otherdyn 7 $\Rightarrow\Rightarrow$ other-1450 otherdyn 4 $\Rightarrow\Rightarrow$ other-40199 otherdyn 4 $\Rightarrow\Rightarrow$ other-15580 otherdyn	2 $\Leftarrow\Leftarrow$ other-47291 https 1 $\Leftarrow\Leftarrow$ other-5101 otherdyn 1 $\Leftarrow\Leftarrow$ other-16768 https 1 $\Leftarrow\Leftarrow$ http otherdyn
Top 5 (histogram by protocol-type of trigger) for **Site** $\mathbf{S}_2$	4 $\Rightarrow$ http http 3 $\Rightarrow$ other-17171 otherdyn 3 $\Rightarrow$ http otherdyn 2 $\Rightarrow$ other-2747 otherdyn 1 $\Rightarrow$ priv-548 otherdyn	4 $\Leftarrow$ other-1250 otherdyn 2 $\Leftarrow$ https printer 2 $\Leftarrow$ http printer 1 $\Leftarrow$ other-2738 otherdyn 1 $\Leftarrow$ other-40060 otherdyn	5 $\Rightarrow\Rightarrow$ http http 4 $\Rightarrow\Rightarrow$ other-65090 otherdyn 3 $\Rightarrow\Rightarrow$ other-54045 otherdyn 3 $\Rightarrow\Rightarrow$ http smtp 2 $\Rightarrow\Rightarrow$ ssh smtp	3 $\Leftarrow\Leftarrow$ other-1340 otherdyn 1 $\Leftarrow\Leftarrow$ other-9100 otherdyn 1 $\Leftarrow\Leftarrow$ other-18428 https

Table 6: Triggers Detected at S_1 and S_2 (using last 5 weeks of D_1 and 4^{th} week of D_2)

trigger-type white-list works best for *(in,multiple)* triggers since it deals with application-level proxies. Note that the remote host heuristic assumes that the remote host is itself not compromised; our results show that even if this white-list is disabled, the number of reported triggers may be manageable.

See Appendix C for examples of the kind of legitimate behavior captured by each of the white-lists.

Validation: Table 6 presents results over our validation datasets using the list of regexps and parameters obtained by analyzing the calibration datasets. After running our algorithm, we first categorized the triggers by type and then clustered them by protocol. In this table, "otherdyn" denotes a set of dynamic ports above 1024. We manually analyzed the Top 5 and Bottom 5 clusters using confidential access to the non-anonymized IP addresses, the full connection logs for those addresses, and in some cases URLs used in Web sessions.

Somewhat discouragingly, we did not find any confirmed attacks. This does not seem due to limitations of our algorithms, however; for reference, we investigated 3 past security incidents from the record of attacks at S_2 and found that our algorithm found at least one trigger involved in each of them. In addition, we found several unusual incidents at both S_1 and S_2 deemed worthy of administrator attention: 2 triggers each at S_1 and S_2 that may represent malicious activity (unfortunately, these could not be confirmed), and a number of unknown relays, P2P applications running on arbitrary ports, and scans, per the following examples.

For S_1, all 4 (out,single) triggers occurred due to a single host's use of a P2P application on non-standard ports. We found that all of the (in,single) triggers we investigated related to a single scan where a (friendly) remote host contacted numerous ports. Some hosts were running services (*e.g.,* on ports 2200 and 32774) rarely accessed. Thus, our training was unable to learn about these services, but the scan "lit them up" simultaneously; the simultaneous occurrences of these connections were indeed causally related, though not because the first enabled the second. For (out,multiple) triggers, we found that all were associated with P2P protocols (*e.g.,* Skype), with the exception of one trigger, "$\Rightarrow\Rightarrow$ https other-48723", which remains a puzzle, as the other-48723 connection attempt indeed appears to have come out of the blue, and we do not know why it was accepted. All 5 (in,multiple) triggers were associated with uncommon P2P applications and instant messaging services. We also found an unknown service corresponding to the trigger "$\Leftarrow\Leftarrow$ other-16768 https".

At S_2, we again found that most of the triggers corresponded to rarely used applications or P2P applications running on non-standard ports. We also found triggers corresponding to possibly malicious activities, port-scanning attempts, and a machine relaying Yahoo web pages for unknown reasons. We also encountered 10 false positives for various reasons, primarily coincidental close-by connections, though some due to data reduction errors in the logs supplied by the sites, which caused us to infer incorrect session structure.

Out of 17 (out,single) S_2 triggers, 4 were inbound HTTP triggering outbound HTTP back to the same host. These are quite suspicious, but the connections were very short and did not include URLs. Another possible attack we unearthed relates to a group of triggers of the form "$\Rightarrow$ other-2747 other-1118" and "$\Rightarrow$ other-2738 other-1109", involving a single local host and a set of remote hosts in China. Apart from these, we found triggers related to the AppleTalk file transfer protocol, Skype, and a data reduction error. In the (in,single) case, we found triggers related to incoming scanning (the same effect as for S_1) and a trigger "$\Leftarrow$ http other-16080" that occurred due to a host that runs its Web server on both 80/tcp and 16080/tcp. We also found a trigger due to an unusual scientific application, and one false positive.

More of the (out,multiple) triggers turned out to be false positives, but we also found a Wiki server where incoming requests to the server evidently cause it to follow a bunch of links outbound. We also discovered hosts running Skype, one unusual trigger of the form "$\Rightarrow\Rightarrow$ other-4251 otherdyn" which may have been an attack, and a messaging application. Out of the 4 (in,multiple) triggers, 3 relate to an AOL client, the other, "$\Leftarrow\Leftarrow$ other-18428 https", was the host relaying Yahoo web pages mentioned above.

In summary, although our results from this analysis are mixed, our algorithm identified 4 possible attacks, with our ability to conclusively label them as attacks being limited because we do not have detailed packet-level traces. In addition, our algorithms unearthed a number of nuggets of administratively useful information. That we can glean these out of a background of millions of connections is encouraging. The monitoring load on the analyst for screening the list of triggers detected by our algorithm appears manageable: 551 triggers over 35 days at S_1, of which more than 400 were caused by a single event (the remote scan), and 72 triggers (again, some due to the same event) over 7 days at S_2. Given these tractable detection levels, we plan to soon operate our mechanism on an ongoing basis at both S_1 and S_2.

8 Implementation

Since our mechanism relies on only connection-level information, it does not require payload analysis, and can therefore be implemented with low overhead. On a contemporary desktop machine (2 Ghz Pentium, 768 MB RAM), our implementation can process 20,000 connections per second, corresponding to an overhead of 50 μsec/connection. The average processing time over a day-long trace from S_1 and S_2 is around 7 and 135 seconds respectively. This overhead is low enough (and can likely be optimized further) that in all but very high-volume environments, our scheme can be implemented as part of a real-time network intrusion detection system monitoring a site's access link.

We drive our implementation from connection-level information produced by the Bro network intrusion detection system [17]. We implemented both offline and online modes. In the offline mode, we process the ASCII connection-level logs generated by running Bro at a site's access link. Our online implementation registers with Bro's event engine for notification of pertinent TCP events: connection establishment, teardown, or timeout (no data activity for some duration of time).

8.1 Pseudo-Code

Our description of the algorithm so far has been in two different logical modules, trigger detection and trigger differentiation. Our implementation folds these two functions together, and is essentially a greedy approach. It first tries to account for the connection C using the database of regexps and a list of recent sessions; if that is not possible, we then exercise our probability test to check if it is a triggered connection.

Algorithm 1 gives pseudo-code corresponding to our online implementation. We use the notation S to refer to a session, C for a connection, and (S, C) for a session S' consisting of the connection C appended to S.

The algorithm first performs session-level parsing (Steps 4-13). We check (Steps 4-8) if a new connection C can possibly be part of a legitimate session by looking for an open session S such that (S, C) is a valid prefix of some session regexp. It not, we then determine (Steps 9-14) whether C is part of an open aggregate session. If the algorithm reaches Step 16, then C is not part of a legitimate session, at which point we run the probability test, testing C against every ongoing session S within

Algorithm 1 Online Implementation

1: State per local host: List of sessions (within the last $\max(\mathsf{T}_{session}, \mathsf{T}_{trigger})$ secs), rates per session type over the last T_{rate} secs, white-lists of trusted trigger types, trusted remote hosts, trusted ports
2: Read in new connection C involving local host L
3: {*check if C matches a known regexp*}
4: **for all** open sessions S involving L within last $\mathsf{T}_{session}$ secs (in decreasing order of arrival time) **do**
5: **if** (S, C) is a prefix of a sequence that matches a regexp and passes the $\mathsf{T}_{session}$ timing test **then**
6: Add C to S, go to Step 2
7: **end if**
8: **end for**
9: {*check if C is part of an aggregate session* }
10: Find most recent open session S that satisfies two conditions: same type as C and involves same remote host as C. Let C' = most recent connection in S of the same type as C.
11: **if** (C', C) are less than T_{aggreg} secs apart **then**
12: add C to S, go to Step 2
13: **end if**
14: If less than T_{rate} seconds have elapsed since the start of monitoring, skip attack check since rates may have not stabilized yet, and go to Step 29
15: {*check if C is correlated to an older session by using our statistical test*}
16: **for all** open sessions S involving L within last $\mathsf{T}_{trigger}$ secs (in decreasing order of arrival time) **do**
17: **if** P[T(C),T(S),interarrival-time)] $< \alpha$ **then**
18: **if** (S,C) satisfies some regular expression **then**
19: add C to S, go to Step 2
20: **else**
21: **if** [(S,C) does not match any of the white-lists] **then**
22: Flag (S,C) as an unexpected trigger
23: Mark C as triggered, go to Step 2
24: **end if**
25: **end if**
26: **end if**
27: **end for**
28: {*C has no correlation to any open sessions*}
29: Initialize a new session from C, add it to the list of open sessions, update rates, update white-lists, go to Step 2

$\mathsf{T}_{trigger}$ seconds. If we determine that (S, C) fails the statistical test, then we further perform two tests. First, since we typically set $\mathsf{T}_{trigger}$ to be larger than $\mathsf{T}_{session}$, we perform the regexp check once again; if it passes this check, we add C to S. Second, we also check the pair (S, C) against our white-lists. If both tests fail, we declare (S, C) as an interesting trigger.

In Step 5, we also perform certain application-specific tests to determine whether C can indeed belong to the session S. We do this for two applications: FTP-data connections should fall within the FTP connection that begins the session, and connections belonging to a login session should fall within the SSH/Telnet connection. (For other applications, the connections need not overlap.) In Step 10, in the aggregation test, we check that the most recent session S that has the same type and same remote host as C. For HTTP alone, we allow the session S and connection C to involve different remote hosts, since a single browser session may traverse multiple websites (as reported in [16]).

8.2 Regular Expressions

We now describe how we implemented the regexp operations required by our algorithm. Although we have been using the term regexps when discussing dynamic port numbers and host identities, these do not strictly correspond to the pure definition of regular expressions, which disallow any context. To incorporate context such as "second connection involves the same remote host as the first", we use regular expressions augmented with back-references.

Our algorithm requires two regexp functions, prefix matching (does a string match a prefix of a regexp?) and back-references (for port numbers and remote host identities). These however are very expensive to implement using standard libraries, so we use an optimized home-grown implementation. The basic idea behind making partial matches faster is that we translate the regular expression into a DFA (Deterministic Finite Automaton) by deriving a state transition table. The state stored in a session S now also includes information about the regexp matching: the regexp that it is a prefix of, and the current state in the DFA corresponding to that regexp. Further, we remove all dead states (states from which no accepting state can be reached). This allows us to implement the partial match operation (S, C) in $O(1)$ time: we do a single lookup in the transition table corresponding to the connection C. This assumes that a session can be a prefix of only one regexp. If, on the other

hand, multiple regexps $R_1, R_2, \ldots, R_n$ potentially have some sessions in common, then we simply add a new regexp $R' = R_1|R_2|\ldots|R_m$ and remove the individual regexps. We implement back-references by maintaining additional context in every session for supporting remote host identities and dynamic port numbers.

9 Discussion

The main design philosophy behind our trigger detection mechanism is to infer causality "from a distance," trading off exact detection (such as an operating system could ensure) for statistical detection. When combined with our trigger differentiation mechanism, our results demonstrate the feasibility of leveraging causality analysis for identifying malicious traffic without any aid from the end-host. Our scheme, however, can be evaded by attackers who ensure that their attack connections occur far apart in time; this is a fundamental limitation when observing only temporal characteristics.

An attractive feature of our solution is that we can use it as a general tool for detecting causality. This gives us a powerful principle that we can apply in several contexts, such as identification of peer-to-peer applications, unauthorized relays, or worms.

The problem of peer-to-peer traffic identification has become increasingly difficult as peer-to-peer applications endeavor to escape policy restrictions. An alternative to payload analysis [21], which may be too expensive in some contexts, is to leverage the fact that the search/download protocol in several peer-to-peer networks involves several connections that are causally related. From our preliminary experiments, it appears feasible to use our mechanism to detect such traffic.

The problem of attackers misusing machines for relaying spam is becoming more widespread. Often the way this works is that a spammer sends a spam message to a pre-established backdoor on a zombie machine, which then relays it to hundreds of recipients. We can use our analysis to identify the causal link between the incoming SMTP connection and the outgoing SMTP connections. Finally, others have used the notion of causality to detect worm attacks by modifying the end-host [7, 9, 15]. We can use our mechanism to identify such causality at the network level by correlating the incoming connection that infects a new victim with the victim's subsequent outbound scans as the worm attempts to propagate further.

We note that our mechanisms for inferring triggers and for differentiating triggers are decoupled. If desired, we can use the latter along with other trigger inference mechanisms (say, using end-host instrumentation). Our mechanisms do not perform payload inspection, which greatly reduces the network processing burden, and also means that the mechanisms apply equally well to encrypted traffic. On the other hand, to improve the false positive performance, we could consider complementing our algorithms with a packet content analysis mechanism, where we use the latter to perform a more detailed assessment of suspect connections that have been flagged. Finally, the ability of our trigger detection mechanism to ferret out application session structure is likely of independent interest to the network measurement community.

10 Conclusion

We have demonstrated the feasibility of inferring likely causal relationships between a host's network connections based solely on analyzing the timing structure of these connections. We do so by modeling network session arrivals as Poisson processes and testing for violations of the null hypothesis that the arrival time of a newly observed connection is independent of that of any existing sessions. This detection framework allows us to determine expected false positive rates, assuming the Poisson assumption holds.

We then set about determining which of the many causally-related connections are actually of interest in terms of indicating unexpected and possibly malicious activity. This requires codifying the session structure associated with different Internet applications as well as leveraging notions of implicit trust (e.g., if the local host initiated contact to the remote host, then the remote host is less likely to be attacking the local host) and past behavior (via training). These techniques succeed with a minimal amount of manual site-specific configuration.

Trace-driven assessments of our mechanisms indicate that they work quite well, both in detecting triggers and session structure, and in winnowing down these detections to those of actual interest. For a good-sized site we find false positive rates on the order of 1–2 dozen/day, which appear to be low enough that an analyst may find the level tolerable.

As part of future work, we are interested in automating the process of inferring regexps of application sessions from traces. We are also interested in adapting our general mechanisms for detecting causality in specific settings such as identifying peer-to-peer traffic.

References

[1] ANDERSON, LUNT, JAVITS, TAMARU, AND VALDES. Detecting unusual program behavior using the statistical components of NIDES. *SRI International's System Design Laboratory* (May 1995).

[2] BARBARA, D., WU, N., AND JAJODIA, S. Detecting novel network intrusions using bayes estimators. In *Proc. SIAM International Conference on Data Mining* (April 2001).

[3] BARFORD, P., KLINE, J., PLONKA, D., AND RON, A. A signal analysis of network traffic anomalies. In *Proc. ACM SIGCOMM Workshop on Internet measurment* (NY, USA, 2002).

[4] CERT Advisory: Increased Activity Targeting Windows Shares. `http://www.cert.org/advisories/CA-2003-08.html` .

[5] CERT Incident Note: Attacks Using Various RPC Services. `http://www.cert.org/incident_notes/IN-99-04.html` .

[6] CERT Advisory: Exploitation of Vulnerabilities in Microsoft RPC Interface. `http://www.cert.org/advisories/CA-2003-19.html` .

[7] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., SHANNON, C., AND BROWN, J. Can we contain Internet worms? In *Proc. HOTNETS* (2004).

[8] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference* (San Antonio, Texas, Jan 1998), pp. 63–78.

[9] CRANDALL, J. R., AND CHONG, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. International Symposium on Microarchitecture* (Portland, Oregon, December 2004).

[10] JAVITZ, H. S., AND VALDES, A. The SRI IDES statistical anomaly detector. In *Proc. IEEE Symposium on Research in Security and Privacy* (1991), pp. 316–326.

[11] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *Proc. International Conference on World Wide Web* (2002).

[12] JUNG, J., PAXSON, V., BERGER, A. W., AND BALAKRISHNAN, H. Fast portscan detection using sequential hypothesis testing. In *Proc. of the IEEE Symposium on Security and Privacy* (May 2004).

[13] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *Proc. ACM/USENIX Internet Measurement Conference* (Miami,Florida, October 2003).

[14] MAHONEY, M. V., AND CHAN, P. K. Learning nonstationary models of normal network traffic for detecting novel attacks. In *Proc. ACM SIGKDD international conference on Knowledge discovery and data mining* (NY, USA, 2002).

[15] NEWSOME, J., AND SONG, D. Dynamic taint analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS* (Feb 2005).

[16] NUZMAN, C., SANIEE, I., SWELDENS, W., AND WEISS, A. A compound model for TCP connection arrivals for LAN and WAN applications. *Computer Networks 40*, 3 (2002), 319–337.

[17] PAXSON, V. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999) 31*, 23–24 (1999), 2435–2463.

[18] PAXSON, V., AND FLOYD, S. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking 3*, 3 (1995), 226–244.

[19] ROESCH, M. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference* (1999).

[20] ROSS, S. M. *Introduction to Probability Models, 8th Edition*. Academic Press, 2003.

[21] SEN, S., SPATSCHECK, O., AND WANG, D. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proc. 13th International Conference on World Wide Web* (2004).

[22] STANIFORD, S., CHEUNG, S., CRAWFORD, R., DILGER, M., FRANK, J., HOAGLAND, J., LEVITT, K., WEE, C., YIP, R., AND ZERKLE, D. GrIDS – A graph-based intrusion detection system for large networks. In *Proc. National Information Systems Security Conference* (1996).

[23] UPPULURI, P., AND SEKAR, R. Experiences with specification-based intrusion detection. In *Proc. Recent Advances in Intrusion Detection* (2001).

[24] WARRENDER, C., FORREST, S., AND PEARLMUTTER, B. A. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy* (1999), pp. 133–145.

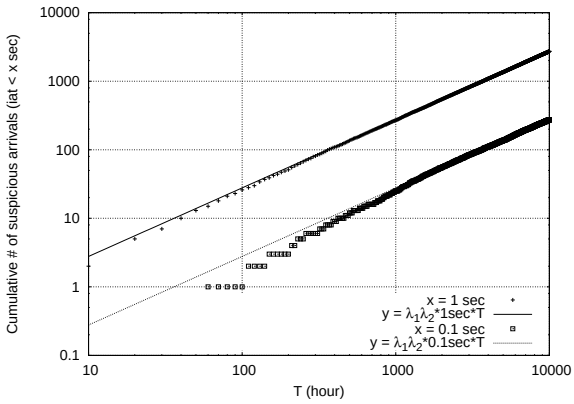
[25] XIE, Y., SEKAR, V., MALTZ, D., REITER, M., AND ZHANG, H. Worm Origin Identification Using Random Moonwalks. In *Proc. IEEE Security and Privacy (to appear)* (Oakland, CA, May 2005).

[26] ZHANG, Y., AND PAXSON, V. Detecting Stepping Stones. In *Proc. 9th USENIX Security Symposium* (Denver, CO, Aug 2000).

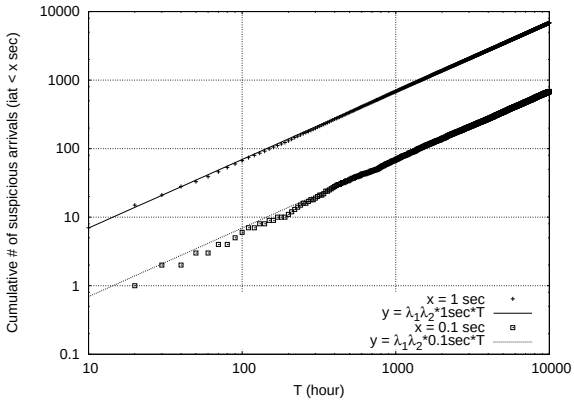
A Monte Carlo Validation Of Lemma 2

We validated our formula for false positives in trigger detection using Monte Carlo simulation. We simulated two Poisson arrival processes, A and B, and counted the number of unusual events where an event from A came within an interval x of an event from B. We generate random numbers, A_i and B_i, representing interarrival times for these two processes. Both A_i and B_i follow the exponential distribution with two sets of rates, either $\lambda_1 = 100/\text{hr}, \lambda_2 = 10/\text{hr}$ or $\lambda_1 = \lambda_2 = 50/\text{hr}$. The first models the case where the triggered and triggering connections have widely different rates, and the second models the case where both process have the same rates. For each set of rates, we generate arrivals until 10,000 hours have lapsed. We then assess each arrival to determine whether it would qualify as "suspicious," defined as follows. A pair of successive arrivals is suspicious if: (a) The first one comes from arrival process A and the second one from B. (b) The interarrival time is less than x seconds. For each experiment, we count the number of suspicious events for $x \in \{1, 0.1, 0.01, 0.001\}$ per every 10 hours. We repeat each experiment with 10 different pseudo-random seeds and average over all 10 experiments.

Figure 4 compares the number of suspicious pairs predicted using our model $\lambda_1\lambda_2 Tx$ (from Lemma 2) with that measured from our simulation, as we increase T. We see that for both sets of rates the values converge quite quickly. (Note that both axes are log-scaled.)



(a) λ_1= 100/hour and λ_2= 10/hour



(b) $\lambda_1 = 50/\text{hour}$ and $\lambda_2 = 50/\text{hour}$

Figure 4: Number of events where type I and type II connections arrive within 1 second and within 0.1 second

B Triggers Detected Prior to Differentiation

This section presents details of the "pure" triggers found using our trigger detection mechanism in isolation, *i.e., without* using our trigger differentiation mechanism. The main conclusion is that network traffic is replete with many different forms of triggered connections, and thus it is vital to also use a differentiation mechanism to distinguish which of these are of operational interest.

Table 7 presents the results of running our trigger detection mechanism tuned to use $\alpha = 0.01$. To do so, we run on the validation datasets rather than the calibration datasets, as discussed in Section 5. The tables classify triggers according to their protocol-type, where the notation "otherdyn" denotes a set of dynamic ports above 1024. An examination of a subset of these triggers indicates that most of them are indeed legitimate. Consider, for example, the 1,043 "$\Rightarrow$ ftp ftp-data" triggers for S_1 that arise due to the fact that the *ftp* application generates several correlated connections. Similarly, the "$\Leftarrow$ http http" triggers for S_2 are due to the fact that a normal browser session involves multiple *http* conections. The 192 "$\Rightarrow\Rightarrow$ other-8080 http" triggers for S_2 are of a different nature: unlike the other triggers, which occur for legitimate reasons at several hosts, these occurred at only a single host (a proxy). The same applies to the "$\Rightarrow\Rightarrow$ other-8088 https" and "$\Rightarrow\Rightarrow$ other-5101 https" triggers.

C Triggers Filtered by White-Lists

Table 8 illustrates the kind of legitimate behavior captured by our white-lists. For each site we show the Top 5 triggers matched by different types of white-lists. Con-

	(out,single)	(in,single)	(out,multiple)	(in,multiple)
Count	1,221	105,008	11,774	2,625
Top 5 (histogram by protocol-type of trigger) for **Site S_1**	1,043 ⇒ ftp ftp-data 107 ⇒ ssh ident 48 ⇒ smtp ident 6 ⇒ ftp-data ftp-data 4 ⇒ ssh ssh	47,871 ⇐ ssh ssh 22,483 ⇐ http http 4,662 ⇐ other-8080 other-8080 3,382 ⇐ other-2200 otherdyn 2,181 ⇐ other-2401 otherdyn	10,722 ⇒⇒ ssh pm_dump 136 ⇒⇒ ssh http 119 ⇒⇒ smtp smtp 86 ⇒⇒ other-36352 otherdyn 78 ⇒⇒ ssh smtp	848 ⇐⇐ http http 216 ⇐⇐ smtp smtp 128 ⇐⇐ other-1023 other-1023 67 ⇐⇐ ssh ssh 62 ⇐⇐ other-60344 otherdyn
Top 5 (histogram by protocol-type of trigger) for **Site S_2**	221 ⇒ ftp ftp-data 47 ⇒ ftp otherdyn 6 ⇒ http ident 5 ⇒ other-5900 http 4 ⇒ other-4822 otherdyn	22,627 ⇐ http http 11,048 ⇐ https https 4,029 ⇐ ssh ssh 675 ⇐ other-6171 otherdyn 490 ⇐ priv-993 priv-993	192 ⇒⇒ other-8088 http 146 ⇒⇒ other-8088 https 35 ⇒⇒ http http 30 ⇒⇒ other-5101 http 30 ⇒⇒ http https	3,337 ⇐⇐ http http 321 ⇐⇐ time time 57 ⇐⇐ https https 46 ⇐⇐ other-40000 otherdyn 38 ⇐⇐ other-8080 other-8080

Table 7: Undifferentiated Triggers detected at S_1 and S_2 (last 6 weeks of D_1, last 5 million conn. of D_2)

White-list Used	*Trusted Trigger Types*	*Trusted Remotes*	*Trusted Ports*
Top 5 (histogram by protocol-type of trigger) for **Site S_1**	142 ⇐⇐ http http 32 ⇐⇐ smtp smtp 19 ⇒⇒ smtp smtp 14 ⇐⇐ other-54045 otherdyn 9 ⇐ other-60344 https	21 ⇒⇒ smtp smtp 4 ⇒⇒ other-993 smtp 4 ⇒⇒ imap4 smtp 3 ⇒⇒ ssh smtp 3 ⇐⇐ other-873 other-873	143 ⇐⇐ http http 33 ⇐⇐ smtp smtp 14 ⇐⇐ other-54045 otherdyn 12 ⇐ other-60344 https 7 ⇐⇐ other-60344 otherdyn
Top 5 (histogram by protocol-type of trigger) for **Site S_2**	11,076 ⇐⇐ http http 2,008 ⇐ http ssh 1,304 ⇐⇐ time time 990 ⇐ other-6171 otherdyn 640 ⇐ other-39281 otherdyn	990 ⇐ other-6171 otherdyn 640 ⇐ other-39281 otherdyn 315 ⇒⇒ http https 122 ⇐⇐ other-39281 otherdyn 93 ⇒⇒ other-8088 http	11,155 ⇐⇐ http http 2,009 ⇐ http ssh 1,304 ⇐⇐ time time 990 ⇐ other-6171 otherdyn 640 ⇐ other-39281 otherdyn

Table 8: Triggers Filtered by White-Lists at Sites S_1 and S_2 (over 2^{nd} weeks of D_1 and D_2)

sider the *Trusted Trigger Types* white-list. It accounts for commonly seen triggers, including multiple access to services on the same host (*e.g.,* 11,076 "⇒⇒ http http" in S_2 triggers) relaying (*e.g.,* 19 "⇒⇒ smtp smtp" triggers in S_1). The *Trusted Remotes* white-list captures legitimate triggers that arise at hosts running unusual applications (*e.g.,* 990 "⇐ $other$-6171 otherdyn" in S_2), and relaying to trusted hosts (*e.g.,* 315 "⇒⇒ http https" in S_2). *Trusted Ports* only captures (in,single) and (in,multiple) triggers—mainly triggers due to repeated use of services, as well as some unknown services that operate on fixed ports.

Even though the triggers captured by these white-lists overlap, there is value in using all of them, as shown in Table 5 in Section 7. This is because each of the four categories of legitimate triggers may arise due to different reasons and are captured by different white-lists.

D Regular Expressions for Common Applications

Table 9 shows the complete list of regular expressions used in our implementation to capture commonly used applications. *dir* is a bound variable used to denote the direction of the corresponding connection; $\overline{dir}$ denotes the opposite direction. We expect that most of these applications—excepting the engineering and database ones which may be specific to our datasets—are likely applicable to other sites as well. We determined these regular expressions by manual analysis of legitimate triggers detected by our algorithm followed by investigation of the applications associated with the ports, and in some cases, payload analysis.

Table 9: Regexps for Characterizing Legitimate Sessions

Application	Regular Expression	Remarks
Common Services		
SMTP	(*dir*,smtp,H) ($\overline{dir}$,ident|finger,H)?	ident/finger used for verifying DNS name
FTP	(*dir*,ftp,H) ($\overline{dir}$,ident|finger,H)? (*in*|*out*,ftp-data,H)*	name verification and data transfer
PortMapper	(*dir*,pm_dump,H) (*dir*,pm_getport,H) ((*dir*,ident,H)|(*dir*,finger,H))? (*dir*,P,H)*	pm_dump and pm_getport offer a directory of services running at host
Login	(*dir*,shell|ssh|telnet|rlogin|exec,H) ($\overline{dir}$,ident|finger,H)? ($\overline{dir}$,priv-X,H) ($\overline{dir}$,X11,H)*	priv-X channel used for control traffic X11 for display redirection
Windows Services		
MicrosoftDirectory	((*dir*,other-3268,H)|(*dir*,ldap,H)* (*dir*,kerberos,H)	name resolution followed by name authentication
NETBIOS	(*dir*,netbios-ssn,H) ((*dir*,priv-445,H) ($\overline{dir}$,priv-445|netbios-ssn,H)) | ($\overline{dir}$,netbios-ssn,H))	secure version of netbios runs on port 445 client may contact both default and secure ports
WINS	(*in*,wins,H) (*out*,wins,H')	wins name resolution may involve relaying
Collaboration Apps		
IRC	(*dir*,irc,H) ($\overline{dir}$,ident,H)? (*dir*,irc,H)*	optional *ident*
AOL	(*out*,aol,H) (*out*,http,H)? (*out*,aol,H)*	redirection to website on chat client startup
NetMeeting	(*dir*,other-1720,H) (*dir*,P,H)*	dynamic ports used for voice traffic
Jabber	(*out*,other-5222|other-5223,H) (*out*,other-5269,H')*	communication between jabber client (5222/5223) and jabber server (5269)
Groove	(*dir*,other-2492,H) ($\overline{dir}$,other-2492,H')*	synchronization between multiple machines
Database Apps		
OracleDBApp	(*dir*,other-1521,H) (*dir*,other-24xx,H)	dynamic port allocation
OracleSQL	(*dir*,other-1525,H) (*dir*,P,H)*	dynamic port allocation
Ariel	(*dir*,priv-419|priv-422,H) (*dir*,P,H)*	dynamic port allocation
PostGresSQL	(*dir*,other-5432,H) (*dir*,P,H)*	dynamic port allocation
MySQL	(*dir*,other-3306,H) (*dir*,P,H)*	dynamic port allocation
TWiki	(*in*,other-9050,H) (*out*,other-3306,H')*	twiki frontend contacting database backend
Engineering Apps		
ProEngineer	(*dir*,other-7788,H) (*dir*,P,H)*	dynamic port allocation
Legato	(*dir*,other-7938,H) (*dir*,P,H)*	dynamic port allocation for license verification
SolidWorks	(*dir*,other-16560,H) (*dir*,P,H)*	dynamic port allocation
CADApp	(*dir*,other-5280,H) ($\overline{dir}$,P,H)*	dynamic port allocation
GridFTP	(*dir*,other-2811,H) ($\overline{dir}$,ident|finger,H)? (*in*|*out*,P,H)*	custom FTP protocol
P2P Apps		
BitTorrent	(*in*|*out*,other-688x,H) (*in*|*out*,P,H')*	search/download protocol
XMule	(*in*|*out*,other-4662|other-4663|other-4664|other-4665,H)* (*in*|*out*,P,H')	search/download protocol
Misc Apps		
RTSP	(*dir*,priv-554,H) (*dir*,P,H)*	dynamic port allocation for media
Quake	(*dir*,other-27000,H) (*dir*,P,H)*	dynamic port allocation for game traffic
WinSock	(*dir*,other-5022,H) (*dir*,P,H)*	winsock relaying