

**Characterizing and Improving Distributed Network-based Intrusion
Detection Systems (NIDS): Timestamp Synchronization and Sampled Traffic**

By

ELLIOT PARKER PROEBSTEL
B.S. (Simmons College) 2004

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Matt Bishop (Chair)

Associate Professor Chen-Nee Chuah

Assistant Professor Hao Chen

Committee in Charge

2008

Acknowledgements

I would like to extend my deepest gratitude to my advisor, Matt Bishop. His insight and support have guided me through the past several years. Through his mentorship, I have learned not only technical skills but also political savvy and professional integrity.

I would also like to thank Chen-Nee Chuah for her academic guidance and support. Her patience with this “term project that never ended” has been nearly infinite, and I sincerely appreciate her willingness to continue giving input and feedback on my work for so long.

This work certainly would not have been possible without the financial support of Sandia National Laboratories, who funded my work through the Sandia National Laboratories/University of California at Davis Excellence in Engineering Fellowship. Moreover, I would like to give personal thanks to Steve Hurd of Sandia for his professional mentorship and for his support in helping me find my career focus.

Countless others have helped guide and support this work in large and small ways. I am unable to convey the depth of my gratitude to Aria Stewart for her boundless inspiration and support – and for always patiently helping me learn how to dig myself out of technical holes. I would also like to thank Jianning Mai of UC Davis and Tao Ye of Sprint for their assistance on the NIDS sampling work, Jerome Braun of UC Davis for his invaluable statistics tutoring, and Hao Chen of UC Davis for enthusiastically agreeing to serve on my thesis committee.

Additionally, I want to acknowledge my dataset sources, CRAWDDAD and CAIDA. Support for OC48 data collection is provided by DARPA, NSF, DHS, Cisco and CAIDA members.

Finally, I would like to thank my parents for their unwavering support as I traversed this often-rocky path. Their faith in my abilities has often exceeded my own and has been a source of strength more often than they may ever know.

Table of Contents

Chapter 1: Introduction	1
1.1 Outline	2
Chapter 2: Timestamp Synchronization – Introduction.....	4
Chapter 3: Timestamp Synchronization – Related Work	7
Chapter 4: Timestamp Synchronization – Possible Approaches	9
4.1 Linear Regression.....	9
4.2 Data Filtering.....	14
4.3 Resistant Regression.....	19
Chapter 5: Timestamp Synchronization – Observations and Results.....	21
5.1 Linear Regression Results	23
5.2. Data Filtering.....	24
5.3 Resistant Regression.....	26
Chapter 6: Timestamp Synchronization – Practical Limitations of Resistant Regression	28
6.1 Scale	28
6.2 Drawbacks to Resistant Regression.....	29
Chapter 7: Timestamp Synchronization – Future Work.....	30
Chapter 8: Timestamp Synchronization – Summary	31
Chapter 9: NIDS Sampling – Introduction	32
9.1 Background.....	32
9.2 Purpose	33
Chapter 10: NIDS Sampling – Related Work.....	34
Chapter 11: NIDS Sampling – Approach	35
11.1 Methods	35
11.1.1 Network Trace Information	35
11.1.2 Sampling Methodology.....	36
11.1.3 Intrusion Detection System Description	36
11.1.4 Determining Consistency Rates.....	37
11.2 Assumptions and definitions	38
Chapter 12: NIDS Sampling – Observations and Results: SNORT	41
12.1 Low Sense Level	41
12.2 Medium Sense Level	42
12.3 High Sense Level.....	42
12.4 Discussion.....	43
Chapter 13: NIDS Sampling – Observations and Results: Bro	45
13.1 Scan Analyzer.....	45
13.2 TRW-Scan Analyzer	47
13.3 All Scans.....	49
Chapter 14: NIDS Sampling – Data Set Analysis	51
14.1 Full Traces	51

14.2 Sampled Data Sets	51
14.2.1 Packet Count	51
14.2 Flow Count	52
Chapter 15: NIDS Sampling – Future Work.....	58
Chapter 16: NIDS Sampling – Summary.....	60
Chapter 17: Conclusions	62
Appendix A. SNORT Results.....	65
A.1 SNORT low sense level.....	65
A.2 SNORT medium sense level.....	67
A.3 SNORT high sense level.....	69
Appendix B. Bro Results	71
B.1 All Scans	71
B.2 Scan Analyzer	72
B.3 TRW-Scan Analyzer.....	73
Appendix C: NIDS Sampling Glossary	75
Appendix D. Code for determining SNORT consistency.....	76

Chapter 1: Introduction

Intrusion detection system (IDS) implementations have increasingly become an integral tool for security practitioners – in particular, network systems analysts, who commonly employ network-based IDSs (NIDS) at the edge of local networks. By automating the task of observing network traffic and detecting, logging, and – in some cases – responding to, malicious traffic patterns, IDSs alleviate the burden of watching network borders for would-be attackers.

In the best case scenario, a NIDS will accurately and consistently monitor bidirectional traffic, producing logs of observed events and, depending on configuration, possibly raising real-time alerts and/or implementing automated responses to anomalous traffic. Under less ideal circumstances, a NIDS may fail to produce alerts for malicious traffic, raise excessive alerts for more benign traffic, or provide an analyst with feedback that the analyst cannot understand or cannot use.

This work marries the results of two independent projects with a united goal of characterizing and improving distributed intrusion detection systems – especially those systems most likely used by network administrators with limited resources and under the constraints that such administrators often face. The first project is the Timestamp Synchronization project. It addresses the issue faced by an administrator who finds, in reviewing aggregated event logs after a network event (such as a network break-in or system compromise), that the timestamps of the log are corrupted due to lack of synchronization between the distributed hosts generating the alarms. Building on the thesis work of Thomas Ristenpart [19], we offer a heuristic approach for an analyst to correct those timestamps to reflect a unified view of time using only the data already

present in the event logs themselves. This contribution should be especially valuable for post-event analysis in circumstances where compromised systems are no longer available for data gathering, or where the compromises are so thorough that the systems cannot be trusted to produce accurate post-event feedback.

The second project is the NIDS Sampling project, which provides guidance to network analysts who are unable to implement a NIDS on full network traces, likely because of resource constraints. As high speed network access continues to grow not only in popularity but also in both bandwidth and throughput, many practitioners find that running a NIDS on all network traffic is simply too resource-intensive. A resulting solution is often to perform traffic sampling and to feed the sampled traffic into the NIDS. Because many NIDS software packages were written to analyze network traffic in the context of surrounding traffic, we demonstrate that sampled packets are often an insufficient data source for a NIDS to perform accurate traffic analysis.

1.1 Outline

In Chapter 2, we formally introduce the timestamp synchronization problem and the major concepts relating to it. Chapter 3 provides an overview of past related work on time synchronization. Chapters 4 and 5 present several possible approaches for addressing the issue and evaluate those approaches using empirical data; in particular, resistant regression is highlighted as being especially successful, especially when applied to minimally filtered data sets. Chapter 6 highlights some possible drawbacks to our approaches and potential limitations of the work, while chapters 7 and 8 provide guidance for future work and a brief summary.

Chapter 9 introduces the problem space for the NIDS sampling project. Chapter

10 reviews related work and, in particular, explains the ties we have explicitly worked to preserve between this project and previous work in the field. Chapter 11 discusses our approach, while chapters 12 and 13 present our findings using SNORT and Bro, respectively. In chapter 14, we analyze the data sets used for the NIDS sampling work, and chapters 15 and 16 provide discussions of our findings and suggestions for future work.

Finally, in Chapter 17 we present unifying observations and conclusions from the timestamp synchronization project and the NIDS sampling project. Supporting appendices follow, providing detailed records of our findings.

Chapter 2: Timestamp Synchronization – Introduction

Aggregated event logs, integrating the alerts from distributed network sensors, can provide a holistic view of network events and a consolidated, chronological history of events occurring on as wide of a scale as we distribute the sensors - the span of an enterprise network, for example. However, the assumption that all alerts will arrive in the order in which they occurred is naive and unrealistic. Large and complicated network designs result in sensors at varied physical and logical distances from the aggregation server. Thus, even in optimal network conditions, alerts from one sensor may arrive before alerts from another sensor, even if the alerts from the closer sensor were generated after the alerts from the more distant sensor. Under less optimal network conditions, the effects may be more pronounced. Given that poor network conditions are often the result of the very type of events whose alerts are of interest (malicious intrusion, hardware/software failure, etc.), this is exactly the time frame in which we hope to analyze an accurate aggregate log.

Ideally, we could solve the problem by ensuring that network sensors timestamp their alerts. In the event that doubt arose as to which alert was generated first, consulting the timestamps would resolve the issue. However, this approach is empirically limited by the inherent assumption that all participating network sensors will demonstrate a unified view of time. When this assumption breaks down, so, too, does the approach.

The Network Time Protocol (NTP) was developed specifically to address the problem that clocks, including those on modern computers, are prone to error. Even if occasionally synchronized, clocks experience drift, a phenomena by which their view of time gradually (and linearly) deviates from an objective view of time. If two network

sensors are synchronized to the same clock at a given time, they will gradually stray from this unified view of time. The longer the clocks go between synchronizations, the more pronounced the effect of drift will be. In addition, Paxson's work [18] demonstrates that even well-synchronized clocks are prone to extreme error on occasion, proving that implementing NTP on a network's hosts does not offer a guarantee that the clocks on these hosts will be reliable timekeepers.

Furthermore, there exist some network situations in which NTP cannot be used – for security reasons, for example – or in which it has simply been misconfigured or not enabled. As Kohno observes [11], all of the following operating systems (among others) either do not maintain system time via NTP or do so only infrequently: default Red Hat 9.0, Debian 3.0, FreeBSD 5.2.1, OpenBSD 3.5, Windows 2000 and XP, and Pocket PC 2002. For these reasons, NTP cannot be seen as the summary answer to the issues surrounding timestamp synchronization.

Given that we cannot necessarily trust the ordering of the events in the aggregated log, and given that we cannot necessarily trust the timestamps applied to the event alerts, is there any way we can examine only the log itself and derive an accurate ordering of events with timestamps that reflect a unified view of time? Ristenpart [19] proves that the theoretical answer is no, but provides an empirical approach for approximating a correction of timestamps in order to compensate.

As our work draws strongly from the problem area outlined and addressed by [19], we adopt the nomenclature used in that work and present those definitions:

- *clock skew*: the initial offset from actual time at time zero
- *clock drift*: the rate of change of the clock's skew

Our work was begun to empirically validate the algorithm used in simulation tests of [19]. This algorithm was found to solve the problem of clock skew robustly, but the problem of clock drift was too significant to be assumed negligible – an assumption in Ristenpart’s work. Drift has been proven to be roughly linear under most computing conditions [11, 16, 18, 23]. So, we derive the drift of network sensors in order to apply a correction scheme that is supported by [18]. We factor the skew into the corrected timestamp for an event at a rate that is proportional to the amount of time that has passed between the event at hand and the time of the first event in the log file.

We are thus able to meet our goal: *using only the data available in the original aggregated event log*, we are able to provide an approximation heuristic that provides a more accurate reflection of the ordering and timing of events recorded in the log. We are aware of no other work that has accomplished this goal without relying on bidirectional communication data – such as the round-trip time for packets between two hosts [16, 18].

What follows are an overview and exploration of related work - including an analysis of why no prior approaches suffice to meet the goals outlined by this project, a discussion of possible approaches and their strengths and drawbacks, results of our empirical study to support an approach involving simple data filtering and resistant regression, and some consideration of the limitations to this approach. Our contribution to this field is in demonstrating that it is feasible to approximate the drift of network sensors, relative to the drift of the server, and correct the timestamps of the aggregated log such that all timestamps reflect a unified view of time – all using only the information available in the log itself. This work could enable post-event correction of security logs, for example, in order to aid forensic analysis.

Chapter 3: Timestamp Synchronization – Related Work

The seminal work in this area by Duda et al. [6] proposed to use simple linear regression or convex hull estimates. Duda [6] noted, in particular, “When the variability grows, the regression analysis gives time scaling errors,” and suggested that in such situations, the convex hull estimation method was more appropriate. A more comprehensive discussion of this particular failing of linear regression can be found in Section 4.1. For now, it is sufficient to note that the motivation of the work at hand involves the specific need to be accurate in situations where the network conditions are highly variable – any approach that fails under such conditions is certainly inappropriate for the current undertaking. However, the convex hull method suggested by Duda et al. requires bidirectional communication traces; without it, the method cannot establish the hull within which to work. Given that we only have unidirectional traffic in the aggregated network logs, the convex hull approach is untenable as a solution.

Paxson’s work [18] confirms empirically the limitation in the simulation studies of [6]: simple linear regression is too prone to error to serve as a reliable correction scheme, especially for data containing high variability. Paxson also suggests a novel calibration scheme based loosely on robust line fitting, but this scheme requires not only bidirectional traffic but also *tcpdump* traces from both agents in a network transaction, rendering it infeasible for our work.

Moon, Skelley, and Towsley [16] assert that it is not possible to derive clock skew (which they term “offset”) from logs of one-way communication – noting that it is impossible to extract transmission and propagation delay times from unidirectional traffic logs – and propose a linear programming solution to compete with [18] on similar data

sets. Our work accepts the fundamental impossibility in detecting delay times, which agrees with Ristenpart's work [19] outlining the theoretical constraints within which his approximation heuristics – and, by extension, ours – function.

More recently, Kohno, Broido, and claffy [11] implemented Moon's algorithm [16] to identify remote physical devices by fingerprinting these devices based on their clock drift rates. Kohno demonstrates that it is possible to use a remote system to detect and map clock drift on a variety of devices running an array of operating systems [11]. This work, though designed for a different purpose, provides further motivation for ours by supporting some of our underlying claims: clock drifts on modern computers are non-trivial, detectable, and generally constant over time.

Chapter 4: Timestamp Synchronization – Possible Approaches

A number of approaches could be used to address the problem of correcting post-event aggregated logs. We first discuss several options and their limitations – some of which are fundamental, and some of which are practical. In keeping with the overall goals of this project, we have attempted to find solutions that are relatively simple to implement and which should be feasible for a system administrator with constrained resources. For this reason – and others, such as ease of use – all statistical work for this project was done using the open source R Statistical Package.

Our evaluations were conducted on data sets we gathered on a campus honeynet. Our logs were generated using the simple case of two sensor hosts communicating log data to a single log server. We used syslog logging capability, but we used a custom script to wrap syslog messages before they were transmitted via UDP to the log server in order to prevent the server from automatically stripping sensor host timestamps from log entries – which it otherwise would have done. Many, but not all, of the events observed by a sensor host were also observed by the other host; this contributed to network congestion at times of high activity on the honeypot. In Chapter 5, we document the result of applying to our log data the approaches we outline here. The results of this applied work are consistent with our expectations of performance, which is promising.

4.1 Linear Regression

Under ideal circumstances, our sensor host h would generate an alert and immediately transmit it to the server with a constant propagation time p . The server would timestamp the alert with arrival time s , and for every pair of timestamps (h, s) in our aggregated

event log we would have:

$$h + p = s$$

Then, for entries $I-i$: $\{(h_1, s_1), (h_2, s_2), (h_3, s_3), \dots, (h_i, s_i)\}$ in the aggregated log, we have:

$$h_i + p = s_i$$

However, in practice, we do not find that our alerts experience a constant propagation time from h to s , and moreover we can neither trust our clocks to be synchronized nor to remain in lockstep over the course of the log file generation. Rather, the effects of *clock skew* and *clock drift* become quite pronounced. This yields something more like the following:

$$h = k + (1 + \delta) s + \varepsilon$$

where k is the *clock skew*, δ is the *clock drift*, and ε is the variation in propagation time. With some minor assumptions on ε , this form becomes the standard linear regression model, which attempts to produce a single line that minimizes the square of the distance between the line and every data point on the plot.

We gather a set of log entries $I-i$ with timestamps h_1, \dots, h_i from the sensor host and corresponding timestamps s_1, \dots, s_i from the server to obtain an estimate of k as the intercept and calculate an approximation of the drift via:

$$\hat{\delta} = \hat{\beta} - 1$$

where $\hat{\beta}$ is the estimated slope coefficient. Because 1 is a constant term, we can use the estimated slope coefficient directly for $\hat{\delta}$.

Ideally, we would like to calculate drift on all relevant systems by comparing their observations of time with a single standard clock, such as an NTP time server. Our aim for this project, however, is to achieve a unified view of time for all entries in an

aggregated log file without requiring access to any information not already present in the file itself. We build on the logic used by Ristenpart [19], where one sensor host is selected as a reference host and *clock skews* are calculated for all other sensor hosts to adjust for the estimated difference in observed time at the start of the log file. However, we cannot directly calculate *clock drift* as a comparison between various sensor hosts, as we have no way of using log file contents to infer differences in those hosts' views of time. What we do instead is regress each host against the log server. With perfect linear drift on all systems and no variation in the error term, our data would look (in a very simplified fashion) like this:

$$(h_1, s_1), (h_2, s_2), \dots, (h_{101}, s_{100}), (h_{102}, s_{101}), \dots, (h_{202}, s_{200}) \dots$$

In this example, it is clear that host h is experiencing a *clock drift* rate of $+1/100$ by comparison to the observed view of time at server s . That is, for every 100 seconds that s observes in passing, h observes the passing of 101 seconds. Assume that the same log file also contained entries from host g that demonstrated the following:

$$(g_1, s_1), (g_2, s_2), \dots, (g_{102}, s_{100}), (g_{103}, s_{101}), \dots, (g_{204}, s_{200}) \dots$$

Here, host g is experiencing a *clock drift* rate of $+2/100$ (or $+1/50$) by comparison to the observed view of time at s . In the time that it takes for server s to advance 100 seconds, g advances by 102 seconds. Although we have no log entries that directly match timestamps from g to timestamps from h , we can apply our knowledge of their individual comparisons to s . Note that at instant 1 , both h and g were perfectly synchronized with s (or, at least, as perfectly synchronized as our tool's granularity allowed us to measure), so we won't need to address *clock skew* and can focus exclusively on *clock drift*. Thus, if we wanted to select host h as our reference host, we could calculate the difference

between the *clock drift* of h and the *clock drift* of g by:

$$(\text{drift}[h] - \text{drift}[g]) = (\text{drift}[s] - \text{drift}[g]) - (\text{drift}[s] - \text{drift}[h])$$

We know the values of the two phrases on the right hand of the equation:

$$(\text{drift}[s] - \text{drift}[g]) = -1/50 = -.02$$

$$(\text{drift}[s] - \text{drift}[h]) = -1/100 = -.01$$

This resolves our simple example:

$$(\text{drift}[h] - \text{drift}[g]) = (-.02) - (-.01) = -.01$$

allowing us to say that host g has a clock that is operating 1% slower than the clock at host h . If we want to correct timestamps from host g so that they are consistent with host h 's view of time, we should adjust each entry downward by subtracting 1% of the period of time g believes has passed since the last time we knew h and g to be synchronized. In our example, this was time instant 1 . So the entry (g_{102}, s_{100}) would be corrected to form g_{102}' :

$$g_{102}' = 102 + (-.01 * 102) = 100.98$$

Because our tools operate with a granularity of seconds, g_{102}' would be set to 101. This does, in fact, yield it consistent with h . Similarly, g_{204}' would be calculated at 201.96 and rounded to 202, again yielding the expected consistency. Thus, if our log files experienced perfect linear *clock drift* and no variance in the error term, linear regression (which is only a slightly more complicate version of what we've presented in this simple example – and which would yield the same results) would suffice.

We must note, however, that the estimated standard error of the estimated slope coefficient will be given by:

$$s^2 \{\hat{\beta}\} = \frac{MSE}{\sum_{i=1}^n (t_i - \bar{t})^2}$$

where MSE is the mean square error, an estimator of the variance of the random propagation errors, and \bar{t} is the mean of the t_i entries. Hence, the estimated error of the drift estimator will depend on the number of entries i in the aggregated log, the variance of the propagation times, and the distribution of actual times. Thus, it is not guaranteed to have any particular desired behavior and will be strongly dependent on the accuracy of all measurements and minimization of errors.

Standard linear regression (the method of Least-Squares Analysis) is known to be most effective on data sets where the error term demonstrates *homoskedasticity* – that is, where random variables in the data set demonstrate a fixed finite variance. When all (or most) data points in the set are known to fall within a football shape when plotted, the data set can be said to be *homoskedastic*; this set will be a good candidate for linear regression. Unfortunately, the presence of errors in our measurements – where by “errors” we mean variations in propagation times, fluctuations in processor load, or any number of unpredicted variables that influence the recorded timestamps – is quite nearly guaranteed. Our data sets are known to be *heteroskedastic*, meaning that the variance among data points is not fixed. A discussion of some unexpected variables that manifested in our data sets is available in Section 4.2, where we also present some ad-hoc solutions for addressing a few of the error patterns we encountered.

For now, we note that the detrimental impact of *heteroskedasticity* in linear regression limits its usefulness in our experiments. Standard linear regression is known to be highly susceptible to large variation in efficacy under the presence of “noise”, or outlier data points that are not consistent with the rest of the dataset. A few outliers can

be sufficient to skew the results of a linear regression application out of consistency with the rest of the data set. We demonstrate an empirical example of this limitation in Section 5.1, where we applied an adaptation of standard linear regression. Ultimately, this drawback to standard linear regression made it insufficient for our work.

4.2 Data Filtering

One way of addressing the limitations of linear regression would be data filtering, a process by which an analyst manually (or perhaps eventually through an automated script) removes the most obvious “noise” from the dataset. In some cases, this could be quite easy, despite there not being foundational statistics theory to support this approach.

Statistics, as a field, is often concerned with identifying the source of outlier data and explaining its presence. In our case, we know the general source of outlier data: variable network conditions and processor loads generate variations in propagation and alert processing times. Network links become congested; sensor hosts experience varying amounts of processor load, delaying their administration of security alerts; network hardware backlogs and queues packets for eventual delivery. Especially when a network is under attack, we expect these errors to manifest in any number of error conditions. This simplifies our work. An analyst need only identify areas where backlog or congestion seem to have influenced the dataset and remove those outlier data points. Given that we expect our data points to nearly approximate a linear pattern, it should often be easy to identify outliers for removal.

In our data set, one set of errors came from the log server being able to timestamp only one alert per second, while our host sensors were able to produce and timestamp multiple alerts per second. Thus, we found many instances where we expected the

timestamps on a series of alerts to look like this:

$$(1_h, 3_s), (2_h, 4_s), (3_h, 5_s), (4_h, 6_s), (5_h, 7_s), (6_h, 8_s)$$

or maybe this:

$$(1_h, 3_s), (2_h, 4_s), (2_h, 4_s), (2_h, 4_s), (4_h, 6_s), (5_h, 7_s)$$

where $(1_h, 3_s)$ represents a timestamp pair from a log entry, reflecting a timestamp from moment 1 at host h and a timestamp from moment 3 at server s . Because of the short time frame in the above windows, we expect the impact of *clock drift* to be negligible – allowing us to represent the relation (for this window of time) between h_i and s_j simply as:

$$h_i + p = s_j$$

Instead, due to the queuing at the log server, the series of alerts were found to skew thusly:

$$(1_h, 3_s), (2_h, 4_s), (2_h, 5_s), (2_h, 6_s), (2_h, 7_s), (3_h, 8_s), (3_h, 9_s)$$

Because the sensor host was able to generate alerts more frequently than the log server was able to receive and process them, the difference in observed time – the difference between the log server’s timestamp and the sensor host’s timestamp on the same alert – would occasionally magnify rapidly. As shown above, in a period of just a few seconds (as observed by the sensor host), the mapping from i to j (for h_i and s_j) grew from $\{i = j + 2\}$ to $\{i = j + 6\}$.

We developed a *trimming* procedure for addressing this particular source of noise. It worked reasonably well, as is demonstrated in Section 5.2, where we show the results of our various approaches. The trimming procedure worked by scanning a log file for any series of entries where the timestamp from the sensor host was identical. The

algorithm simply cut all but the first of such entries, guaranteeing that for any given host sensor timestamp h_i , there existed only one log entry marked with timestamp h_i . This approach provided a straightforward method of trimming out some of the erroneous entries, but it was not sufficient to remove all entries thusly affected. For example, note again the example series:

$$(1_{h,3_s}), (2_{h,4_s}), (2_{h,5_s}), (2_{h,6_s}), (2_{h,7_s}), (3_{h,8_s}), (3_{h,9_s})$$

After our trimming procedure, the same series would be trimmed to this:

$$(1_{h,3_s}), (2_{h,4_s}), (3_{h,8_s})$$

The problem here is that although we managed to remove four erroneous entries, we are still left with the entry $(3_{h,8_s})$, which demonstrates an exaggerated skew between h and s . The more pronounced these bottlenecks are, and the longer they last, the more of an impact they can have on our results. Using standard linear regression, even very occasional instances of erroneous data points can be sufficient to distort our results.

Moreover, the bottlenecks could be less visible from an individual sensor host perspective if, for example, a number of sensors were all experiencing high traffic at the same time instant. This could result in a temporarily exaggerated skew for these hosts, yet there would be no series:

$$(1_{h,3_s}), (2_{h,4_s}), (2_{h,5_s}), (2_{h,6_s}), (2_{h,7_s}), (3_{h,8_s}), (3_{h,9_s})$$

from which to directly trim. Rather, imagine a simplified case of distributed sensors e, f, g , and h , where the non-bottlenecked traffic looked like this:

$$(5_h, 7_s), (6_g, 8_s), (7_h, 9_s), (10_e, 12_s), (12_f, 14_s), (13_h, 15_s), (15_e, 17_s)$$

In that case, we assume a non-variable propagation time, and we assume that e, f, g , and h are synchronized and experience no drift. Even with this highly simplified case, the

introduction of a bottleneck might skew the log timestamps so that we see this:

$$(5_h, 7_s), (6_g, 8_s), (6_h, 9_s), (6_e, 10_s), (6_f, 11_s), (7_h, 12_s), (7_e, 13_s)$$

Here, we cannot prove from the data alone that a bottleneck has occurred, making it difficult to automate detection for such scenarios. We can observe that the propagation time for sensors e, f , and h demonstrated a rapid change in a very short period of time, but there is no clear guide for what to trim. In our simplified case, it could be argued that we should apply the same rule as we discussed in the case of a single-host bottleneck: if any two hosts have entries with the same host sensor timestamp, we could cut them. But what if we add complexity to the model by relaxing our assumptions? For example, if we simply move to a model where we cannot assume that e, f, g , and h are synchronized – even while assuming no drift – then only the third, sixth, and seventh entries in the above example can be proven suspect. We must move to a model where we compare entries from host h to other entries from host h and assume no relationship between timestamps from various hosts.

If we do not have any outside evidence to suggest an explanation for what was occurring at the time of the above example, we could just as reasonably interpret the data to suggest that:

- A bottleneck occurred around 6_s , and all entries time stamped 6_s to 13_s should be adjusted to reflect a near-simultaneous arrival at the log server; or
- Hosts h , and e are experiencing very significant drift with relation to the log server, and their entries should not be adjusted.

Generally, given the context of the rest of the log file, we can often determine which of these two assumptions is most likely to be correct. If the trend observed in the

above example continues throughout the log file, then either we have no perceived baseline from which to draw our estimations of drift, or we can conclude that the very pronounced drift we observe in the example above is a true representation of the drift taking place on hosts h , and e . If the trend from the example does not continue throughout the entire log file, however, we can reasonably assume that the period of rapid exaggeration on drift estimation is an anomaly and can likely be excluded from our estimations.

We did not attempt to manually identify periods that appear evidence of bottlenecks and remove all corresponding entries. It is not difficult to imagine any number of manual trimming or filtering techniques: for example, an analyst could specify a time period during which she knew the network to have been particularly congested and simply chop all entries from that time period. Alternately, if the analyst did not already know the times during which bottlenecks had occurred, she could map p (where p is defined as the difference between the host timestamp and the server timestamp for each timestamp pair) against the time scale as observed by the sensor host to get a quick visual representation of bottlenecks and use this as guidance for what to trim. However, as the example above suggests, identifying periods of erroneous data can quickly become an art rather than a science. In addition, an analyst's ability to identify these periods can depend heavily on the scale chosen for viewing the data. Our data sets tended to appear perfectly *homoskedastic* when viewed at full scale; zooming in on particular time windows was the only way to visually detect the variance in our error term.

We chose to implement only a simple filtering technique that could easily be supported with foundational arguments. We demonstrate in Section 5.2 that this

approach was sufficient to yield very promising results on nearly all of our data sets. We leave it as an area for future research to extend our results with other filtering techniques and note also that developing these techniques will necessarily be specific to the data sets at hand; a technique that proves highly useful on one data set may not solve any issues for another data set that does not demonstrate the same pattern of errors. For example, in a case where sensor hosts and log servers are capable of processing an equal number of alerts per time frame, bottleneck trimming might not be useful and instead might trim out useful, accurate data points.

Even with our chosen trimming procedure, the straightforward application of simple linear regression yields results that are not sufficiently accurate for our use¹. That is, our trimming is not thorough enough to reduce our *heteroskedastic* data set to one that mimics *homoskedasticity*. Though there may be trimming procedures that would render the data sets “clean” enough (that is, free enough of outlier data points) for linear regression to reliably produce accurate results, we chose to explore a more resistant regression method rather than to continue developing trimming methods. Our results showed this choice to be a fruitful one.

4.3 Resistant Regression

Resistant regression was developed specifically for addressing the need to apply regression analysis to data sets known to be *heteroskedastic* or known to contain outliers – as both of these can generate distorted feedback from standard linear regression. Our data sets are expected to both be *heteroskedastic* and to contain a number of outliers, making resistant regression a natural match for our work.

¹ See section 5.2 for empirical evidence and in-depth analysis.

Though there are a few methods of implementing resistant regression, we've chosen to explore the Least Trimmed Squares (LTS) method, as implemented by the function `lqs()` in the *R Statistical Package*. LTS is similar to standard linear regression (least-squares regression), which generates a line that minimizes the squared distance between the chosen line and all data points; these distances are also known as *residuals*. As previously noted, the effort to minimize all residuals makes linear regression very sensitive to outliers. LTS attempts to overcome this limitation by trimming out roughly half of the data points (those with the largest residuals) and performing a least-squares analysis on the remaining points [25]. A common concern with LTS is that it displays such sensitivity to central data values [24]. For our work, this actually turns out to be an advantage, as we want our regression to focus on the data values that display a marked linear trend and not be biased toward outliers.

Chapter 5: Timestamp Synchronization – Observations and Results

We are able to evaluate the accuracy of the various methods proposed above by comparing the results of those methods to the *clock drift* we measured on the three computers used for our research: Host 1 (*H1*), Host 2 (*H2*), and the log Server (*S*). In order to determine this baseline, we compared the system times on *H1*, *H2*, and *S* with the time on a NIST NTP server, noted the differences, and then performed this comparison again roughly three days later. We computed the clock drift for each system:

$$\frac{(N_2 - h_2) - (N_1 - h_1)}{(N_2 - N_1)}$$

where N_i represents the time at the NTP server at time i , and h_i represents the time at the host (or log server) at time i . Though we would preferred to have gathered more than two data points on our systems, and to have gathered them more than three days apart, we were constrained by a series of system issues, which occurred shortly after the second NTP comparison and which gave us reason to doubt that future experimentation on these systems would yield consistent results. We are confident, however, that the data we were able to gather before the system issues is accurate and valid.

Using the results from the comparisons with NTP servers, we established the following drift rates:

H1: 7.7750 e-06 seconds/second

H2: 2.0059 e-05 seconds/second

S: 3.5236 e-05 seconds/second

Note that all three drift rates are positive, meaning that all three of our systems were

drifting forward in their view of time, with S advancing most quickly and $H1$ advancing least quickly. Since we are always correcting our log files to be consistent with either $H1$ or $H2$, we note:

$$H2 - H1 = (S - H1) - (S - H2) = 2.0059e-05 - 7.7750e-06 = 1.2284e-05$$

If we select $H1$ as our reference host, we will need to subtract from $H2$ at a rate of $1.2284e-05$; conversely, if $H2$ is selected as the reference, we will need to add to $H1$ at a rate of $1.2284e-05$.

Our event logs were generated between 29 January 2006 and 19 February 2006. We pulled six logs from this timeframe, some of which have overlap with one another. In total, the logs contained 2063 entries; roughly 30% of those entries were from $H1$, while the other 70% were generated by $H2$.

As shown in Table 4.1, we used event logs covering varying spans of time and with varying numbers of entries in order to test our approaches on a heterogeneous set of log files. The *time span* column indicates the difference between the first and last timestamps in the file, rounded to the nearest hour. Logs A-F are arranged chronologically, where log A reports the earliest events and log F reports the most recent.

Table 5.1 Log file characteristics.

Log File	Time Span	$H1$ Entries	$H2$ Entries	Total Entries
A	79 hours	64	146	210
B	165 hours	148	490	638
C	166 hours	155	296	451
D	82 hours	23	92	115
E	164 hours	99	201	300
F	162 hours	113	236	349

5.1 Linear Regression Results

We first applied linear regression directly to a prepped version of our data sets², performing all work on a Dell Inspiron 4100 laptop with an Intel Celeron 1GHz processor with 256MB of RAM. In *R*, linear regression is the standard function `lm()`. To demonstrate, consider a case where we have a `.csv` file called `logA1.csv` containing two columns, labeled `ht` and `st`, for *host time* (from *H1*) and *server time*. In the following code snippet, we first read in the log file and store it in the object `logA1`. We then perform linear regression on `logA1` by regressing `ht` against `st` and store the results in object `lmA1`. Finally, we calculate the *clock drift* for *H1*, as compared to *S*, and store the result in the object `driftA1`³.

```
> logA1 <- read.csv("logA1.csv", header=T)
> lmA1 <- lm(ht ~ st, data=logA1)
> driftA1 <- coef(lmA1)[2]-1
```

We repeat this process for `logA2` to generate the difference between the *clock drift* of *S* and the *clock drift* of *H2*, storing the result in the object `driftA2`. The difference between `driftA1` and `driftA2` should be (if our data sets were homoskedastic and contained no errors) the difference in *clock drift* between *H1* and *H2*. By repeating this for all six log files, we should be able to verify our results – assuming, again, that our log files represented homoskedastic data sets with few or no outliers. As shown in Table 4.2,

² The preparation involved stripping all information from the syslog entries and splitting each log file into two files, one for each host. Thus, log A became log A1 and log A2, where log A1 contained only entries from *H1*, and log A2 contained only entries from *H2*, etc. We also selected the earliest timestamp in each log and set it as time 0, adjusting all other timestamps in the log as offsets from 0. This made the processing work in *R* more straightforward, though it certainly could have been done without this advance work.

³ Note that we need to subtract 1 from the value generated by the linear regression, because `lm()` has calculated the slope of the line, which is necessarily the value of the drift plus 1. That is, if the host was found to have no drift whatsoever, linear regression would generate a flat line with a slope of 1. We subtract this baseline value to acquire the true drift value.

our results from $lm()$ were far from ideal.

Table 4.2 Results of linear regression on unfiltered data.

Log File	H1 Drift	H2 Drift	Difference
A	1.1150E-05	1.6564E-04	-1.5449E-04
B	-2.1223E-05	-2.1973E-05	7.5000E-07
C	1.7499E-05	6.7624E-06	1.0737E-05
D	3.9206E-05	1.1159E-05	2.8047E-05
E	2.6919E-05	1.4901E-05	1.2018E-05
F	2.9871E-05	1.5116E-05	1.4755E-05
Avg	1.7237E-05	3.1934E-05	-1.4697E-05

The middle two columns represent the *clock drift* differences between the sensor hosts and the log server – the results of $(\text{drift}[S] - \text{drift}[H1])$ and $(\text{drift}[S] - \text{drift}[H2])$, respectively. Ideally, we should see *H1* Drift column values of approximately $2.7461\text{e-}05$ and *H2* Drift column values approximating $1.5177\text{e-}05$. Only the last two rows, representing log files E and F, demonstrate values within a reasonable range of what we expect. Clearly, direct application of linear regression to our data sets does not provide trustworthy output. This is not cause for concern, however, as we expected linear regression to fail on our data, given that it has a significant number of outliers and given that the error term variance was not fixed.

5.2. Data Filtering

As described in Section 4.2, we chose to implement only a simple data filtering procedure. This short script passed through the data sets and generated new log files by trimming out all but the first log entry for all entries with the same host timestamp. That is, for a set of entries: $\{(1_h, 3_s), (1_h, 4_s), (1_h, 5_s), (1_h, 6_s), \dots, (1_h, n_s)\}$, only $(1_h, 3_s)$ will be preserved, and the rest are discarded. In this way, we minimized the impact of bottlenecks, while recognizing that we did not solve for them entirely.

Table 4.3 Results of data filtering.

Log File	Entries Cut From H1	H1 Entries Remaining	Entries Cut From H2	H2 Entries Remaining	Total Entries Cut From H1 and H2	Total Remaining Entries
A	18	46	45	101	63	147
B	67	81	211	279	278	360
C	36	119	70	226	106	345
D	6	17	10	82	16	99
E	28	71	26	175	54	246
F	19	94	19	217	38	311

We show in Table 4.3 how many entries were cut during the filtering procedure. It is worth noting here that a much higher percentage of entries were cut from the log files which performed badly with linear regression.

After the filtering, we ran the trimmed log files through the same linear regression technique and mapped the new results:

Table 4.4 Results of linear regression on filtered data.

Log File	H1 Drift	H2 Drift	Difference
A	3.3370E-05	2.2026E-04	-1.8689E-04
B	1.6947E-05	2.7957E-05	-1.1010E-05
C	2.2817E-05	3.4998E-06	1.9317E-05
D	3.4244E-05	1.2205E-05	2.2039E-05
E	2.6957E-05	1.5263E-05	1.1694E-05
F	2.8230E-05	1.5158E-05	1.3072E-05
Avg	2.7094E-05	4.9057E-05	-2.1963E-05

We again see that linear regression is not resistant enough to the effect of the remaining outliers, as the results are still not anywhere near close enough to being acceptable. It is possible that more manual trimming techniques might bring the data sets closer to working with linear regression. However, as we show in the next section, the approach of manually trimming outlier data points is unnecessary. Moreover, as this trimming is a manual process, it is slow and cannot be theoretically supported as a robust approach to the problem domain.

5.3 Resistant Regression

For our implementation of resistant regression, we use the function `lqs()`. Using the same structure as the example in section 5.1:

```
> logA1 <- read.csv("logA1.csv", header=T)
> lqsA1 <- lqs(ht ~ st, data=logA1)
> driftA1 <- coef(lqsA1)[2]-1
```

Again, we repeat this process for `logA2` to derive the drift from *H2* in log file A, and then we calculate the difference between the two drift values. Table 4.5 presents the results of applying `lqs()` to the unfiltered data sets.

Table 4.5 Results of resistant regression on unfiltered data.

Log File	<i>H1</i> Drift	<i>H2</i> Drift	Difference
A	2.2633E-05	1.6840E-05	5.7930E-06
B	2.5312E-05	1.4782E-05	1.0530E-05
C	2.6892E-05	1.3459E-05	1.3433E-05
D	6.1144E-05	1.1919E-05	4.9225E-05
E	2.9229E-05	1.4747E-05	1.4482E-05
F	2.6506E-05	1.5069E-05	1.1437E-05
Avg	3.1953E-05	1.4469E-05	1.7483E-05

In the results presented in Table 4.5, we can see the vast difference in accuracy between standard linear regression and resistant regression methods for our data. Even on the raw, unfiltered data sets, the resistant regression approaches accuracy for four of the six log files. The shorter log files (as logs A and D were both gathered over about half the time span as the other log files and contained approximately one half to one third as many log entries) demonstrate the least accuracy. The other four log files (B, C, E, and F) actually average to an *H1* drift of 2.6985E-05, an *H2* drift of 1.4514E-05, and a difference of 1.2471E-05.

We next applied `lqs()` to the filtered data set and achieved the most accurate

results of our testing. Table 4.6 presents the results:

Table 4.6 Results of resistant regression on filtered data.

Log File	<i>H1</i> Drift	<i>H2</i> Drift	Difference
A	2.1423E-05	1.8449E-05	2.9740E-06
B	2.5075E-05	1.3040E-05	1.2035E-05
C	2.9009E-05	1.3361E-05	1.5648E-05
D	3.1898E-05	1.4864E-05	1.7034E-05
E	2.8441E-05	1.4484E-05	1.3957E-05
F	2.7045E-05	1.5098E-05	1.1947E-05
Avg	2.7149E-05	1.4883E-05	1.2266E-05

Using resistant regression on our trimmed data set, we achieve results very close to ideal. Recall that we measured the drift of *H1* as 2.7461e-05 and *H2* as 1.5177e-05, with our calculated difference as 1.2284e-05.

Chapter 6: Timestamp Synchronization – Practical Limitations of Resistant Regression

Our approach has several limitations; here, we discuss the most prevalent. Most notably, we warn the reader about issues of scale and about the known drawbacks to resistant regression.

6.1 Scale

There may be issues with calculating and applying a single drift factor to a log file with a long time scale, i.e. with a large difference between the first and last timestamps. We did not have an opportunity to perform experiments of varying time scales to measure the impact of, for example, calculating the drift on a log file of a shorter duration and applying that drift factor to a log file gathered over a longer time frame. Though our work was designed in such a manner that practitioners could use the same log files for calculating and applying drift factors, it would not be a stretch to imagine that some analysts would extend this work to use a “less tainted” log file to calculate the drift factor and apply this drift value to another log file. We caution that we have not tested this approach and thus make no claims about the applicability of our work to that scenario.

It seems reasonable to believe that such an extension of our work would be a natural move and could work tidily; however, analysts would need to ensure that they were not, for example, calculating drift values on a log file gathered over a span of just a few days and then applying that drift value to a log file spanning several months. The accuracy of this work seems best suited for calculating and applying drift values to the same time scale. However, our work also shows that the results of the resistant regression are most reliable when they are taken from longer log files with a greater

number of “known good” entries. Thus, an intuitive extension of our work might be to gather log files under fixed network conditions and also under variable network conditions and test to see if the drift values calculated on the “fixed” set would apply readily to the “variable” set.

6.2 Drawbacks to Resistant Regression

One of the most frequently cited drawbacks to resistant regression is the exaggerated importance it gives to central values in a data set, those with the least residuals. However, we’ve shown that this is a strength for our application of resistant regression methods.

Another drawback that is often noted for resistant regression is that it is computationally intensive. Our data sets were small enough that we did not experience any delays from the intensiveness of the $\text{lqs}()$ function, but we note the concern here, because it may present obstacles for others who wish to apply our work to their data sets. As mentioned in Section 5.1, all of our work in the *R Statistical Package* was performed on a Dell Inspiron 4100 laptop with an Intel Celeron 1GHz processor with 256MB of RAM – not a workhorse, by any means – but our data sets were quite small. It is possible that larger data sets on similar systems might experience delays that we did not face.

Chapter 7: Timestamp Synchronization – Future Work

There are several directions for future work that would flow naturally from this project. As mentioned in Section 6.1, we have not tested the application of this work in log files of varying time scales. Tests of this nature would be very useful for answering questions like: How many entries are necessary in a log file in order to generate useful results? How high can the *signal:noise* ratio be without affecting the results of the resistant regression analysis? Can the clock drift value calculated from a short log file be applied, without detrimental impact, to a longer log file?

Another direction that we would like to explore is a study of various common logging services to see what types of variance are often experienced. Though bottlenecks were common in our experiment, other logging services might demonstrate markedly different sources of data variance or outliers. Solutions for addressing other error sources might not be as easy to devise – or as easy to automate – as were the bottlenecks in our data. We felt confident that filtering out bottleneck data entries was defensible; other error patterns might not have such obvious and easily supported solutions.

Additionally, our logging service selection dictated our network protocol usage: UDP. Given that UDP is connectionless, entries generated by our log hosts either reached the log server right away or never reached it at all. We performed only a cursory analysis to compare the entries from the hosts themselves to the entries in the aggregated log files. This quick analysis did not immediately draw our attention to any patterns of missing entries. However, it is possible that some entries were lost in transmission and that using a logging service with connection-oriented data transfer might change the nature of our data set patterns. We recommend further exploration in this direction.

Chapter 8: Timestamp Synchronization – Summary

Our work is the first of which we are aware to offer experimental data supporting an approach to deriving correctional values to adjust timestamp values in aggregated event logs. Other work in this field has all concluded, or operated under the assumption, that work of this nature would require bidirectional data traces. Thus, we are the first to produce an approach that suggests it is possible to derive *clock drift* values using only the data inherent to the log files themselves. Paired with Ristenpart's work [19], our drift values can be used in tandem with Ristenpart's *clock skew* values to correct log file timestamps to reflect a unified view of time.

Chapter 9: NIDS Sampling – Introduction⁴

9.1 Background

The primary goal of the network intrusion detection system (NIDS) sampling project was to measure NIDS efficacy in a statistically-sampled environment. Our work was aimed at understanding how well different NIDS installations would work, using a dataset of network traffic captured at different locations on a university campus network and an independent dataset of network traffic captured on an OC48 backbone link. This would allow us to identify which NIDS implementations work best with specific types of traffic (e.g. academic, residence halls, etc.). Furthermore, we would probabilistically sample the network traffic and reassess the accuracy of the NIDS packages in order to identify any shortcomings with traditional intrusion detection when using probabilistically-sampled network traffic.

This work was motivated by a desire to establish a baseline understanding of the accuracy issues faced by enterprise network system administrators, many of whom work under resource constraints that necessitate tradeoffs between capturing all packets (in many cases, not technically feasible) and capturing a subset of packets (less desirable for security purposes). We intend to help inform those who need to make such tradeoffs in their network security design. Knowing the impact of these tradeoffs will allow network administrators to more intelligently apply their resources and interpret the output and limitations of their NIDS.

⁴ Preliminary results from this project have been published as a Sandia National Laboratories Report [17].

9.2 Purpose

The purpose of this investigation was to accurately characterize the challenges associated with conducting intrusion detection in a distributed environment. One part of this investigation focused on issues relating to intrusion detection conducted using a statistically sampled subset of all network traffic. While traditional intrusion detection methods rely on analyzing all traffic, increasingly, statistical sampling of network traffic is being used in today's NIDS.

To our knowledge, there have been no studies to date that establish the necessity of this advanced work for application in enterprise networks. That is, there are no published statistics on the accuracy degradation of current state of the art NIDS when these systems are monitoring sampled data rather than full network traces. Furthermore, all known studies on this topic have been performed on backbone network traffic [1, 13, 14], the characteristics of which are often markedly different from those of an enterprise network, such as what might be found on a university campus.

Chapter 10: NIDS Sampling – Related Work

In the last few years, research has brought attention to the problem area of sampled data being used for anomaly detection. Mai et al. open the door with research into the specific application of data sampled from Internet backbone links [13, 14]. Their work suggests directions for future research, indicating that while traditional intrusion detection mechanisms are not well suited for performing anomaly detection on sampled backbone traffic, there are promising alternatives. Brauckhoff et al. extend this work with a case study of the Blaster worm and apply various detection metrics to sampled excerpts of Blaster-infected network traces [1]. All of these look particularly at Internet backbone traffic and explore detection metrics that have not yet been incorporated into open source NIDS software packages.

Most recently, Ishibashi et al. quantitatively evaluate the impact of sampling rate and monitoring granularity on the detectability of anomalies [9]. They use new mechanisms to demonstrate that changes in measurement granularity can assist in anomaly detection on sampled traffic.

No recent studies have evaluated the specific situation faced by current network administrators who must decide whether sampled traffic will suffice for anomaly detection on their own networks. This is the direction we take our research, and we hope it will not only contribute to the research field but also to the practitioners currently in the field.

Chapter 11: NIDS Sampling – Approach

11.1 Methods

The primary method of investigation was to compare results of two prominent open source NIDS software packages: SNORT [20] and Bro [2]. We used multiple data sets, and within each data set, a range of statistical sampling rates, to best gauge performance degradation associated with statistical sampling.

11.1.1 Network Trace Information

We had initially planned upon using network trace information from a variety of sources. Most prominently, we planned to use information captured at a variety of points on the UC Davis campus network. Unfortunately, due to unforeseen administrative delays, we had little choice but to abandon this collection effort. Thus, inquiry was limited to use of pre-existing network trace information. The two sources, CRAWDAD and CAIDA, which represent enterprise and backbone traffic respectively, are described below.

For enterprise traffic, we are using network traces from the CRAWDAD (Community Resource for Archiving Wireless Data at Dartmouth) project at Dartmouth College [12]. These traces were gathered during the fall of 2003 through sniffing of the Dartmouth wireless traffic and thus represent only the traffic of wireless clients, which we consider a limitation of our dataset. However, the CRAWDAD datasets have provided us with an opportunity to compare NIDS consistency on traffic observed in an academic building with traffic seen in residence halls. The traffic we are using was gathered between November 2, 2003 and February 28, 2004 in one academic hall and two residence halls (Residence Hall 100 and Residence Hall 13).

Our backbone traffic dataset is from CAIDA [5], the Cooperative Association for Internet Data Analysis. The traffic was observed on an OC48 link in San Jose. Though the traces we are using represent only a few hours of traffic on this link, they represent over 100 GB of raw network traffic. Both the CRAWDAD and CAIDA datasets are packet header traces that have been sanitized in a consistent (within the dataset), prefix-preserving manner. Thus, they are quite usable for the purposes of this inquiry.

11.1.2 Sampling Methodology

Each dataset is used in its original format as a “full trace”; what portscans⁵ a NIDS detects in the full trace is considered a baseline for consistency on the trace. We then run the same NIDS configuration on network traces in which the traffic has been sampled. We use a random packet sampling technique in which each packet is sampled with a probability of n and discarded with a probability of $1-n$, where n is ranged to support sampling rates of 1:5, 1:10, 1:25, 1:50, 1:75, 1:100, 1:200, 1:500, and 1:1000. For any given trace, the sampled traces are generated independently, meaning that a 1:10 sampled trace is not a strict subset of the corresponding 1:5 sampled trace.

11.1.3 Intrusion Detection System Description

In order to expand the relevance of our research, as well as to provide baselines for comparison between our work and past research on anomaly detection in sampled traffic, we have performed an analysis of the consistency of the sfportscan preprocessor of SNORT [20], an open source NIDS, on both enterprise and backbone traffic⁶. We have

⁵ For the purposes of our research, we use the working definitions of “portscan” used by the two NIDS software packages we are evaluating (SNORT and Bro). Both packages overload the term “portscan” to refer to both portscans and portsweeps. In general, a portscan is understood as a series of attempted connections to a variety of ports on a single host in order to establish what services are running on the scanned host, while a portsweep is understood as a series of attempted connections (usually, but not always, on the same port) to a variety of hosts in order to establish which hosts are running a given service.

⁶ By analyzing the portscan detection algorithm, we build an explicit link between our work and the work

run SNORT at low, medium, and high sense levels on all of our datasets. Aside from disabling detection for all other anomaly categories and changing the sfportscan sense level, we used default settings for the SNORT configuration.

We also analyzed the other open source NIDS software package Bro [2]. Though Bro requires a much more complex installation and configuration process, there are still some settings for Bro which can be considered default – and these include two distinct scanning detection algorithms: the Scan analyzer and the TRW-Scan analyzer. The Scan analyzer uses a straightforward scan detection mechanism of tracking the number of distinct addresses and ports a host attempts to contact and alerting when those variables exceed pre-determined limits [4]. The TRW-Scan analyzer implements the Threshold Random Walk algorithm introduced by Jung [10] and discussed in the development of Mai’s work [13, 14].

For completeness, we note that in order to use Bro on the CAIDA data, we needed to transform the link-layer headers. The CAIDA dataset was gathered on Cisco-HDLC (cHDLC) links, and Bro is not capable of interpreting headers of cHDLC traces. We used a transformation script written by Aria Stewart [22] that removed the cHDLC headers and wrapped the IP packets with forged Ethernet headers⁷.

11.1.4 Determining Consistency Rates

To establish the *consistency rates*⁸ of each NIDS on a particular dataset, we compare alerts raised by that NIDS on the full trace with alerts raised by that NIDS on

of [14] and [13], which explored portscan detection on backbone data using sampled traffic. We hope to expand the applicability of our work by establishing such bridges whenever possible, enabling more direct comparison between our results and the results of other related work.

⁷ This should not affect our results. We also verified that running SNORT on the forged-Ethernet traces produced the same output as running SNORT on the cHDLC traces.

⁸ See the definitions in section 11.2 for clarification on terms.

the associated sampled traces. Every alert from a sampled trace is compared with alerts from the full trace. For SNORT, matches are generously assumed to be present when an alert from the sampled trace has the same relevant IP address, general scan type, and time window (60 seconds for the low sense level, 90 for the medium sense level, and 600 for the high sense level). For Bro, matching is even more generous; a match is considered positive if an alert from the sampled trace has the same relevant IP address, alert type (matching algorithm), and time window of 3600 seconds. This is the default time window for Bro's scan detection algorithms. The full code used to detect matches between SNORT's sampled trace alerts and full trace alerts is provided in Appendix D; parallel code was used for Bro analysis, with relevant variables tuned respectively.

11.2 Assumptions and definitions

A critical assumption in this research is that short of manually analyzing each network trace in full, there is no definitive method of establishing “ground truth” regarding an IDS' accuracy in detecting portscans.

For example, there may be scans that SNORT or Bro fails to detect in the full network traces – false negatives – and there may likewise be alerts raised by SNORT or Bro that relate to legitimate non-scanning network traffic – false positives. However, the wide use of SNORT and Bro in the security community suggest that their accuracy is acceptable, and furthermore, our intention is not necessarily to establish the baseline accuracy of a NIDS. Rather, we concern ourselves with the question of whether a NIDS will perform consistently on full traces and their corresponding sampled traces. Thus, for this study we introduce the following terms, and their definitions:

- *real scan*: a scan detected by SNORT or Bro in a full trace

- ***false scan***: a scan detected by SNORT or Bro in a sampled trace that cannot be correlated to a *real scan*
- ***missed scan***: a scan detected by SNORT or Bro in a full trace that is not detected in a corresponding sampled trace
- ***detected scan***: a scan detected by SNORT or Bro in a sampled trace that can be correlated to a *real scan*
- ***false scan rate***: the percentage of alerts raised by SNORT or Bro on a given sampled trace that are *false scans*; this metric offers insight into how many false alarms a NIDS is raising at a given sampling rate
- ***missed scan rate***: the percentage of *real scans* not detected by SNORT or Bro in a given sampled trace; this metric offers insight into how many actual attacks a NIDS is failing to detect at a given sampling rate
- ***consistency rate***: the percentage of *real scans* detected by SNORT or Bro in a given sampled trace; this metric is the inverse of the *missed scan rate* and is intended to demonstrate the accuracy of a NIDS at a given sampling rate

As a simple case, assume that a NIDS detected five portscans in a particular full trace and then raised only three portscan alerts on the 1:5 sampled trace. We compare the alerts from the sampled trace to the five real scans and find that one alert relates to a *real scan*, while the other two cannot be correlated to any of the *real scans*. We say here that the sampled trace had two *false scans*, one *detected scan*, and four *missed scans*, yielding a *missed scan rate* of 80% (the NIDS failed to detect four of the five *real scans*), a *consistency rate* of 20% (the NIDS detected one of the five *real scans*), and a *false scan rate* of roughly 67% (two of the three alerts were false alarms).

There is one additional note about terminology we would like the reader to note. In some cases, we refer to the “accuracy” or “performance” of a NIDS we are evaluating. In all parts of this project, we are evaluating only the consistency of a NIDS; the appropriation of terms like “accuracy” and “performance” in this work is only for grammatical heterogeneity – that is, to avoid repeating the same phrases and overwhelming the reader with redundancy.

Chapter 12: NIDS Sampling – Observations and Results: SNORT

We have found that SNORT's *consistency rates* differ significantly between sense levels and between datasets. In our analysis, we discuss *scans* generically, including both portscans and portsweeps. It was often the case that a dataset contained a statistically insignificant number of scans of one type but a very large number of scans of the other type. Appendices A and B present full tables documenting the exact observations from our experiments, while this chapter's discussion will report the broad lessons gained from our research and supply a few abbreviated tables for support.

12.1 Low Sense Level

At the low sense level, detection rates drop so sharply that even at a sampling rate of 1:5, the Academic Hall dataset had the highest *consistency rate* of just below 12%, while the *consistency rates* of other network traces were between 1% and 8%. The accompanying *false scan* rate at the 1:5 sampling rate ranged from 4% to over 77%. With a sampling rate of 1:10, all *consistency rates* fell below 3% with *false scan rates* ranging between 5% and 100%.

Table 12.1 Excerpt of CRAWDDAD Academic Hall Portsweeps at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	8221	0	0.00%	8221	0	0.00%	100.00%
1:5	1207	234	19.39%	973	7248	88.16%	11.84%
1:10	235	39	16.60%	196	8025	97.62%	2.38%
1:25	6	0	N/A	6	8215	99.93%	0.07%

12.2 Medium Sense Level

On the medium sense level, detection rates for the backbone traces still dropped sharply to 12% at a 1:5 sampling rate, but *false scan rates* stayed below 5% for all sampling rates, demonstrating a marked improvement over the *false scan rates* we observed in the low sense level. Even more encouraging, the *consistency rate* for the academic network traffic was over 80% at the 1:5 sampling rate and even stayed above 70% for sampling rates of 1:10 and 1:25 – while displaying very low *false scan rates*.

Table 12.2 Excerpt of CRAWDAD Academic Hall Portsweeps at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	35911	0	0.00%	35911	0	0.00%	100.00%
1:5	32621	2965	8.26%	29656	6255	17.42%	82.58%
1:10	27533	367	1.33%	27166	8745	24.35%	75.65%
1:25	25624	38	0.15%	25586	10325	28.75%	71.25%
1:50	19117	11	0.06%	19106	16805	46.80%	53.20%

12.3 High Sense Level

In our testing of the high sense level, we demonstrated again a sharp drop in *consistency rates* for the backbone traffic and the residence hall traffic – both falling to approximately 10% with a 1:5 sampling rate, dipping below 5% with 1:10 sampling, and finally disappearing below 2% for all remaining sampling rates. We did, however, again observe low *false scan rates* for all sampling ranges; these rates stayed in the single digits across all sampling rates. For the academic network traffic, the high sense level displayed the highest resistance to sampling impact. *Consistency rates* were nearly 60% at 1:5, over 30% at 1:10, and stayed in the 15% range for sampling rates ranging from 1:25 to 1:200. *False scan rates* stayed very low for all these ranges, as well.

Table 12.3 Excerpt of CRAWDAD Academic Hall Portsweeps at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	31480	0	0.00%	31480	0	0.00%	100.00%
1:5	19124	704	3.68%	18420	13060	41.49%	58.51%
1:10	10458	252	2.41%	10206	21274	67.58%	32.42%
1:25	6194	32	0.52%	6162	25318	80.43%	19.57%
1:50	5333	10	0.19%	5323	26157	83.09%	16.91%

12.4 Discussion

SNORT configuration instructions recommend that medium and high sense levels be used only with manual sfportscan tuning, as the higher sense levels often generate more false positives [21]. In all cases, we used the default settings for the various sense levels. SNORT's sfportscan low sense level detects scans based only on the number of RST (connection reset) responses a host receives in a given time window [15]. Given that an active benign host is unlikely to contact a large number of unavailable hosts or services, the low sense level is particularly unlikely to generate many true false positives. However, this also makes the low sense level very sensitive to the effects of random packet sampling, which is known to present a strong bias in favor of longer flows and miss a large percentage of short flows [7]. Obviously, a flow consisting only of a SYN (connection request) packet and an RST packet is very short and would easily be missed by random packet sampling, which helps explain why the low sense level experienced such a high percentage of missed scans.

Medium and high sense levels not only expand the time windows for enumerating RST responses, but they also make use of connection counts per host. By tracking connection counts, these sense levels can detect scans launched against firewalled hosts, but they are also much more likely to raise false alarms on highly active benign hosts.

For this reason, the SNORT Reference Manual recommends that an operator review alerts for such false positives and reactively update the sfportscan configuration to ignore such active hosts in the future to avoid clogging alert logs.

As discussed previously, we do not have the capacity to manually verify whether the alerts raised on the full traces are true positives in the sense of ground truth, as the data available to us is only sanitized packet headers. However, our experiments have demonstrated that SNORT does not scale well on sampled data. Alerts raised on sampled data are not a representative sample of the alerts that would be raised on the full network trace from which those samples were extracted. Not only do *consistency rates* suffer tremendously, but the number of *false scans* tends to be very high.

We have also shown SNORT to be most resistant to the impact of sampling when applied against traffic traversing an academic network using a medium sense level and a sampling rate of at least 1:100. SNORT is also capable of performing moderately well on academic traffic using a high sense level and a sampling rate of at least 1:200. SNORT's consistency, at all sampling rates and with all sense levels, degraded rapidly on both backbone traffic and residence hall traffic.

Chapter 13: NIDS Sampling – Observations and Results: Bro

Bro does not have an equivalent to SNORT's sense level settings, but we present here the results of Bro's two scan detection algorithms using their default configurations. The Scan analyzer is similar in its basic construction to the scan detection algorithms used by SNORT, while the TRW-Scan analyzer implements a more complex algorithm which is purported to detect scans more quickly and more accurately than traditional online scan detection algorithms [10]. Both scan detection algorithms built into Bro demonstrated a moderate level of consistency and roughly matched SNORT's best results.

13.1 Scan Analyzer

The Bro Scan analyzer monitors outgoing connections from a host and raises an alert when the host has attempted to connect a pre-determined number of distinct hosts or ports. The time window is very broad at 3600 seconds, especially as compared to SNORT's widest time window, used in the high sense level, of 600 seconds.

The number of hosts/ports that must be contacted before an alert is raised are user-definable; by default, outgoing alerts are raised when 100 hosts are contacted, again when 1000 hosts are contacted, and yet again when 10,000 hosts are contacted. That is, if a system attempts to contact 1500 hosts in under 3600 seconds, two alerts from the Scan analyzer will be raised – the first when 100 hosts have been contacted, and the second when 1000 hosts have been contacted. Incoming alerts are defaulted to 1000 and 10,000. The direction of the scan is determined by a user-defined setting of `is_local_address`; for our work, we did not use this setting.

Though we could have feasibly made an educated guess about which anonymized address space belonged to Dartmouth in the CRAWDAD traces, we did not choose to. To the best of our knowledge, the only difference this setting would have made is to reduce the number of alerts by ignoring hosts which scanned a number of hosts between 100 and 999. There was also no parallel setting for us to use for the CAIDA traces (there is no logical notion of “incoming” or “outgoing” for traffic that is traversing an Internet backbone link), so we chose to leave this at its default: no value. Furthermore, all other Scan analyzer variables were left in their default state.

Table 13.1 Bro Scan analyzer alerts for CRAWDAD Academic Hall data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3068	0	0.00%	3068	0	0.00%	100.00%
1:5	2687	14	0.52%	2673	395	12.87%	87.13%
1:10	1682	18	1.07%	1664	1404	45.76%	54.24%
1:25	1373	0	0.00%	1373	1695	55.25%	44.75%
1:50	1274	12	0.94%	1262	1806	58.87%	41.13%
1:75	1160	69	5.95%	1091	1977	64.44%	35.56%
1:100	810	116	14.32%	694	2374	77.38%	22.62%

Bro’s Scan analyzer demonstrated particularly high consistency on the CRAWDAD Academic Hall data set, as shown in Table 13.1, though *false scan rates* grew quite sharply at 1:75 and 1:100 sampling rates. We note also that at 1:200, Bro’s Scan analyzer did not generate any alerts at all, so Table 13.1 reflects all Bro results for the CRAWDAD Academic Hall data set.

On the CRAWDAD Residence Hall data sets, the *consistency rates* dropped immediately below 30% at the 1:5 sampling rate, stayed just above 10% at 1:10, and then fell quickly to single-digit consistency rates. However, the *false scan rates* for the Residence Hall data sets never reached even 2%, which we consider a promising result.

We note here that the Scan analyzer raised only about 500 alerts on each of the Residence Hall data sets, which reduces our confidence in the statistical accuracy of our consistency evaluation.

As we will show to be a uniform result from both analyzers, Bro's Scan analyzer performed much less favorably on CAIDA data. Though the *consistency rate* neared 60% at 1:5 and remained above 40% for 1:10 sampling rates, *false scan rates* remained above 10% for sampling rates between 1:5 and 1:50. Only at 1:200 did the *false scan rate* really fall below 5%, and at that sampling level the *consistency rate* also dropped to just over 2%.

Table 13.2 Excerpt of Bro Scan analyzer alerts for CAIDA data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	24284	0	0.00%	24284	0	0.00%	100.00%
1:5	17426	2935	16.84%	14491	9793	40.33%	59.67%
1:10	12405	1675	13.50%	10730	13554	55.81%	44.19%
1:25	6489	771	11.88%	5718	18566	76.45%	23.55%
1:50	2310	232	10.04%	2078	22206	91.44%	8.56%
1:75	1335	113	8.46%	1222	23062	94.97%	5.03%

13.2 TRW-Scan Analyzer

Bro's TRW-Scan analyzer was developed as an implementation of the Treshold Random Walk detection algorithm first proposed in 2004 by Jung, Paxson, Berger, and Balakrishnan [10]. The Bro user manual promotes the TRW-Scan analyzer as "more sensitive scan detection" [4].

It was of particular interest in our study that the TRW-Scan analyzer produced a much higher number of alerts (than the generic Scan analyzer) for all but the CRAWDAD Academic Hall data set. Of the 3148 scan alerts raised by Bro on the Academic Hall data, only 28 – or less than 1% – were raised by the TRW-Scan analyzer.

Compare this to the CAIDA data set, where 90% of scan alerts originated from the TRW-Scan analyzer, or to the CRAWDAD Residence Hall data sets, where the TRW-Scan analyzer produced roughly 85% of scan alerts.

Because the Academic Hall data produced only 28 alerts from the TRW-Scan analyzer, we do not provide discussion of consistency rates for this data set. For completeness, we do provide a table summarizing the alert data in Appendix B, where we supply full tabular results for all data sets.

For both Residence Hall data sets, the TRW-Scan analyzer produced high *consistency rates* with *false scan rates* that never reached 1%. This was a notable result, providing hope that the results of Mai et al. [13, 14] – which characterized and improved on the TRW algorithm using backbone data sets – might be likewise promising on enterprise network data sets.

Table 13.3 Excerpt of Bro TRW-Scan alerts for CRAWDAD Residence Hall 13.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3047	0	0.00%	3047	0	0.00%	100.00%
1:5	1839	9	0.49%	1830	1217	39.94%	60.06%
1:10	1062	3	0.28%	1059	1988	65.24%	34.76%
1:25	429	1	0.23%	428	2619	85.95%	14.05%
1:50	224	0	0.00%	224	2823	92.65%	7.35%

Table 13.4 Excerpt of Bro TRW-Scan alerts for CRAWDAD Residence Hall 100.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3054	0	0.00%	3054	0	0.00%	100.00%
1:5	1969	11	0.56%	1958	1096	35.89%	64.11%
1:10	1155	5	0.43%	1150	1904	62.34%	37.66%
1:25	483	1	0.21%	482	2572	84.22%	15.78%
1:50	228	0	0.00%	228	2826	92.53%	7.47%

Bro performance was not quite as favorable on the CAIDA data set. Though

consistency rates were relatively high, *false scan rates* were over 20% for 1:5 and 1:10 sampling, remained roughly 10% or higher through sampling at 1:25 and 1:50, and didn't drop below 5% until 1:100.

Table 13.5 Excerpt of Bro TRW-Scan alerts for CAIDA data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	241542	0	0.00%	241542	0	0.00%	100.00%
1:5	147012	42423	28.86%	104589	136953	56.70%	43.30%
1:10	85017	20116	23.66%	64901	176641	73.13%	26.87%
1:25	40260	6110	15.18%	34150	207392	85.86%	14.14%
1:50	24070	2296	9.54%	21774	219768	90.99%	9.01%

13.3 All Scans

The overall Bro results, combining alerts from both scan analyzers, offers a favorable comparison to SNORT. The results for the CRAWDAD Academic Hall data set demonstrate the highest performance.

Table 13.6 Excerpt of Bro alerts for CRAWDAD Academic Hall scans.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3148	0	0.00%	3148	0	0.00%	100.00%
1:5	2730	18	0.66%	2712	436	13.85%	86.15%
1:10	1700	20	1.18%	1680	1468	46.63%	53.37%
1:25	1379	0	0.00%	1379	1769	56.19%	43.81%
1:50	1277	12	0.94%	1265	1883	59.82%	40.18%
1:75	1160	69	5.95%	1091	2057	65.34%	34.66%
1:100	814	118	14.50%	696	2452	77.89%	22.11%

When applied to the Residence Hall data sets, Bro's consistency still stayed over 50% at the 1:5 rate and only dipped below 10% at the 1:50 sampling. Most promisingly, Bro never produced even a 1% *false scan rate* for the Residence Hall data. On the CAIDA data, Bro fell just below 45% at 1:5 and only ran under 10% at 1:50. However,

the *false scan rates* were quite high: at both 1:5 and 1:10, Bro produced more than 20% *false scans*, and remained above a 5% *false scan rate* until reaching the 1:100 sampling rate.

Chapter 14: NIDS Sampling – Data Set Analysis

14.1 Full Traces

For enterprise traffic, we are using network traces from the CRAWDAD (Community Resource for Archiving Wireless Data at Dartmouth) project at Dartmouth College [12]. These traces were gathered during the fall of 2003 through sniffing of the Dartmouth wireless traffic and thus represent only the traffic of wireless clients, which we consider a limitation of our dataset. However, the CRAWDAD datasets have provided us with an opportunity to compare NIDS consistency on traffic observed in an academic building with traffic seen in residence halls. The traffic we are using was gathered between November 2, 2003 and February 28, 2004 in one academic hall and two residence halls (Residence Hall 100 and Residence Hall 13). The Academic Hall data set represents over 73 GB of data, while the combined Residence Hall data sets are roughly 18 GB.

Our backbone traffic dataset is from CAIDA [5], the Cooperative Association for Internet Data Analysis. The traffic was observed on an OC48 link in San Jose. Though the traces we are using represent only a few hours of traffic on this link, they represent over 100 GB of raw network traffic. Both the CRAWDAD and CAIDA datasets are packet header traces that have been sanitized in a consistent (within the dataset), prefix-preserving manner.

14.2 Sampled Data Sets

14.2.1 Packet Count

We performed some casual tests on our sampling procedures to ensure randomness – that

is, to ensure that our sampling procedures were not propagating any obviously predictable patterns across sampled data sets. The tests confirmed our expectations that smaller data sets (those generated by less frequent sampling rates) were not strict subsets of larger data sets, with the obvious exception that all sampled data sets were strict subsets of the full traces from which they were generated.

Additionally, SNORT alerts helped confirm for us that our sampling procedures were functioning properly. SNORT scan analyzers include an alert for what SNORT calls a FIN Scan; this is an instance where SNORT observes a packet where only the FIN option is set. Because normal protocol usage never generates packets with only the FIN option set, SNORT alerts on all such packets – regardless of traffic context – as instances of a FIN Scan. Though these “scans” were not of particular interest to us in our general line of inquiry, we used their presence as a gauge to ensure that our sampled data sets were properly scaled in relation to the full data set. In all cases, the 1:5 sampled data set of every trace reflected roughly one fifth as many FIN Scan alerts as the full trace, and similarly scaled results were observed for all other sampling rates. This confirms that our sampling procedure transforms our data in a manner that is consistent with generally accepted knowledge in the field: random packet sampling does not distort general packet statistics; it simply scales them by the factor predicted by the sampling rate.

14.2 Flow Count

As described by Duffield [7] and later confirmed by Brauckhoff [1], random packet sampling is expected to distort flow counts, even at low sampling rates. Short flows often dominate network traffic patterns, but these short flows are less likely to be sampled than are longer flows. Higher sampling rates will also often result in flow

splitting, where a single long flow is perceived to be multiple short flows. This occurs when a flow is sampled once (or more than once), and then is subsequently not sampled again until after the time window expires, causing the observer to presume that the flow has ended. When the flow is sampled again beyond the time window, it is presumed a new flow, again distorting flow statistics. In this way, low sampling rates can drive down flow counts, while much higher sampling rates can cause spikes in flow counts, though both effects are distortions on the true nature of the network traffic.

We present a few measures of flow statistics, which we gathered from the Bro connection reports. These reports are automatically generated when we run Bro on a data set, and we excerpt some statistics from the connection reports to provide insight into the nature of the traffic in our data sets, as perceived by Bro. A direct excerpt from one of our connection reports demonstrates the information provided by these reports:

```
1067794144.082180 1.398963 190.84.69.61 215.81.7.255 http 1072 80 tcp 0 16885 S2 L
1067794266.678484 ? 190.84.69.76 28.250.193.69 other 4751 554 tcp ? ? S0 L cc=1
1067794276.573907 0.000000 190.84.175.224 190.84.69.113 other 2151 1035 tcp 0 ? SH L
1067794276.580403 0.000000 190.84.69.113 190.84.175.224 other 1035 2151 tcp 0 ? SH L
1067794282.130409 0.000000 190.84.69.76 28.250.194.35 http 4738 80 tcp 0 ? SH L
1067794285.225640 0.007746 190.84.65.162 190.84.69.67 netbios-ssn 4430 139 tcp 71 ? SH L
cc=1
```

Figure 14.1 Bro connection report excerpt.

The values follow a standard format, as described in the Bro Reference Manual [3]:

```
<start> <duration> <local IP> <remote IP> <service> <local port> <remote port> \
    <protocol> <org bytes sent> <res bytes sent> <state> <flags> <tag>
```

As Figure 14.1 demonstrates, particular values cannot always be derived for a given flow. In those circumstances, Bro fills the field with a question mark. For our statistics, we use the following fields:

- *<duration>*: specifies the time span of the flow
- *<service>*: reports Bro's guess at the application layer; we cannot confirm the

accuracy of these guesses, because we only have packet headers

- *<org bytes sent>*: byte count from the host that originated the connection
- *<res bytes sent>*: byte count from the host that responded to the connection request
- *<state>*: Bro's summary of the connection, as delineated by one of thirteen state options, described in more detail in Table 14.1.

Table 14.1 Bro Connection Report state guide⁹.

State	Description
S0	Connection attempt seen, no reply.
S1	Connection established, not terminated.
SF	Normal establishment and termination.
REJ	Connection attempt rejected.
S2	Connection established and close attempt by originator seen (but no reply from responder).
S3	Connection established and close attempt by responder seen (but no reply from originator).
RSTO	Connection established, originator aborted (sent a RST).
RSTR	Established, responder aborted.
RSTOS0	Originator sent a SYN followed by a RST, we never saw a SYN-ACK from the responder.
RSTRH	Responder sent a SYN ACK followed by a RST, we never saw a SYN from the (purported) originator.
SH	Originator sent a SYN followed by a FIN, we never saw a SYN ACK from the responder (hence the connection was "half" open).
SHR	Responder sent a SYN ACK followed by a FIN, we never saw a SYN from the originator.
OTH	No SYN seen, just midstream traffic (a "partial connection" that was not later closed).

Unsurprisingly, we found that the CAIDA dataset was mostly populated by connections where only half the communication for any given flow was present. Recall that the CAIDA data was gathered on an Internet backbone link, where politics, financial concerns, and sheer practicality often work in tandem to nearly guarantee that most flows

⁹ This table is a minor adaptation of the state summary guide from the Bro Reference Manual [3].

will appear unidirectional. Carriers practice “hot potato” routing, where they pass off communication to another link – usually one owned by the carrier providing service to the destination host. As a result, traffic for a particular flow rarely routes through the same backbone links in both directions. Indeed, the CAIDA dataset connection reports were entirely populated by states S0, RSTOS0, RSTRH, SH, and SHR – all states that indicate unidirectional traffic observation – in addition to a number of unclassified OTH connections.

Because the CRAWDAD datasets were gathered much closer to the “edge” – that is, much closer to the hosts at one end of the flows – we see a much broader distribution of connection states in the CRAWDAD connection reports. As sampling rates drop, the flow state reports distort to reflect the observer’s fragmented view of the network traffic, providing some key insight into the source of NIDS inconsistent behavior.

Table 14.2a Bro connection states for CRAWDAD Academic Hall data, low sampling rates.

State	Full Trace	1:5	1:10	1:25	1:50
S0	36782366	10559121	5521200	2268793	1144514
S1	92970	89347	28163	5181	1330
SF	2473011	35785	7802	1201	265
REJ	159171	11258	3253	566	151
S2	1667	66995	18554	3233	810
S3	10521	75406	21086	3517	880
RSTO	759096	30863	7789	1216	246
RSTR	109239	4130	838	107	25
RSTOS0	123449	240890	131777	55780	28408
RSTRH	64230	52488	29087	12028	6244
SH	342816	513225	290853	123425	63185
SHR	80239	428361	241929	104050	53098
OTH	29727	378986	249135	116502	61609

Table 14.2b Bro connection states for CRAWDAD Academic Hall data, high sampling rates.

State	1:75	1:100	1:200	1:500	1:1000
S0	766375	575589	288991	115579	57452
S1	593	345	102	15	3
SF	126	66	16	3	0
REJ	70	47	9	1	0
S2	374	214	49	9	2
S3	417	230	53	8	2
RSTO	129	76	18	4	2
RSTR	9	10	1	0	0
RSTOS0	19046	14147	7165	2899	1403
RSTRH	4016	3154	1582	627	290
SH	42239	31699	16161	6432	3143
SHR	35720	26791	13603	5462	2684
OTH	41735	31427	16103	6539	3213

Note how rows 2-8 of Table 14.2a-b (which reflect connections in which the observer was able to monitor bi-directional communication) thin much more rapidly than the rows which reflect connections in which only unidirectional communication was observed. Similar patterns emerged in the CRAWDAD Residence Hall datasets, while the CAIDA dataset simply demonstrated this pattern from the start – and continued it at all sampling rates.

We also extracted statistics relating to the duration of flows and the number of bytes per flow. In nearly all cases, Bro reported “?” or 0 more commonly than any other value for each of those variables, skewing our median values. In the CAIDA data, for example, flows at the 1:5 sample level averaged over 6MB outgoing and 2MB incoming, but the median value for outgoing bytes was 0, and the median for incoming was a low 273. We are unsure whether this is an artifact of Bro’s inability to accurately parse the sampled flows, a side effect from using packet headers only, a combination of the two, or some other variable we did not take into consideration.

It is of interest, though, that flows in the CRAWDAD Academic Hall full trace dataset averaged the longest duration, with a mean over 12 seconds and a median of a

half second. This compares to Residence Hall full trace means of 6 seconds and medians of not even 0.1 seconds, and a CAIDA full trace mean of a half second with a median of barely 0.1 second. Moreover, the Academic Hall dataset had median transmissions of roughly 450 bytes in each direction, while the Residence Hall datasets both averaged approximately 350 bytes outgoing and 0 bytes incoming.

In total, these statistics suggest that the placement of the observation point may bear significant impact on the resulting performance of a NIDS that is monitoring sampled traffic. Even at full trace level, the CRAWDAD Residence Hall traces reflect a view of network traffic that Bro interprets as moderately unidirectional. The connection states that suggest a more bidirectional view of traffic thin more rapidly and more significantly on the Residence Hall traffic reports, while the CAIDA reports reflect a unidirectional traffic observation from the start. We posit that this may account for the decreased *consistency rates* on the Residence Hall and CAIDA traces. Because the NIDS is able to maintain a more holistic view of the network traffic on the Academic Hall traces, better consistency is maintained, and fewer resultant errors arise.

This suggests that analysts who are forced to deploy network-based intrusion detection systems on sampled data can minimize the detrimental impacts of doing so by ensuring that their data gathering is done at points where a more bidirectional view of traffic can be ensured. Our work also suggests that more development is urged in areas of NIDS deployment for circumstances where unidirectional traffic views cannot be avoided. Standard open source packages currently do not have consistency rates that make them attractive options for monitoring sampled unidirectional traffic.

Chapter 15: NIDS Sampling – Future Work

In the future, we would like to extend this work to explore a greater variety of network traffic anomalies. That we only researched scanning detection is a limitation of our work, in that many analysts are not even looking for scanning statistics for their networks. In fact, most will simply discard alerts of this nature entirely. However, we chose to pursue this direction specifically because it would allow us to build explicit links between our work and the prior work of Mai et al.[13, 14]. Moreover, it can be argued that detecting volume anomalies is the most foundational task of an intrusion detection system. While it often requires keeping state information (rather than merely checking a packet against signatures), it represents the simplest in multi-packet NIDS actions. For other attacks that are only detected as a result of stateful analysis, it follows that statistical sampling greatly reduces the performance of a NIDS.

One new direction would be to take data sets including known intrusions and run them through a series of NIDS using a variety of sampling rates. As identifying actual intrusions would likely be a more critical task for a NIDS than merely identifying a port scan, this would provide more relevant feedback as to the criticality of the sampling issue.

Along those lines, it would be instructive to have an experienced NIDS analyst who is familiar with the “ground truth” of the underlying dataset examine the NIDS results (given different sampling rates) and identify detections and missed detections, as well as false positives. Due to constraints on privacy, we expect that this call to action can only be answered by organizations whose privacy policies would allow an analyst to examine full packet traces. An ideal future direction for this work would be to use full

network traces with a current NIDS installation. This would answer the issues of preprocessor tuning, configuration settings, and operator unfamiliarity that we faced.

We hope that our work provides motivation for further research into NIDS software that can support sampled network traces. The promising work being done by UC Davis and Sprint Advanced Technology Labs [13, 14] suggests that there are directions for this work that could lead to exciting new IDS developments. However, as our work demonstrates, solutions which work well for enterprise network traffic do not always work as well for backbone traffic – and the reverse is likely true. Thus, industry would do well to embark on similar, but possibly divergent, paths to ensure that NIDS software can keep pace as enterprise network use continues to grow.

Chapter 16: NIDS Sampling – Summary

Simply put, the results strongly indicate that the use of statistical sampling in NIDS usage causes NIDS consistency to deteriorate significantly. Certainly, there are cases where statistically sampled NIDS performance was reasonably good. The example of a CRAWDAD Academic Hall data, at 1:5 sampling rate and medium SNORT sense level, produced fairly accurate alerts (82.58% of the original alerts). Similarly favorable results from Bro indicate that there are network configurations which lend themselves reasonably well to the installation of a NIDS for monitoring sampled traffic. On the other hand, that same 1:5 sampling rate used with low or high SNORT sense levels yielded far less satisfactory results (11.84% and 58.51% respectively). It is notable that a 1:5 sampling rate is considerably less than the 1:1000 sampling which appears to be commonly used.

From these results, it is clear that NIDS performance is very sensitive to tuning issues. While a NIDS analyst may not be able to dictate the sampling rates, they can adapt other parameters (such as the sense level) to match the needs of their environment.

As mentioned in section 2.1, it is regrettable that we were unable to secure UC Davis campus information for this analysis. However, even had we used that information for the analysis, it seems unlikely that the results and conclusion would vary significantly from those presented.

Finally, as mentioned earlier, this research is based upon the assumption that the 1:1 sampling rate detections represent “ground truth”. While it seems intuitive that 1:1 sampling provides the highest fidelity results, it is possible that some or all of the alerts generated through 1:1 sampling, but not generated at higher sampling rates, represent

false positives (in terms of ground truth). This is a thorny issue and is the core problem of NIDS research. NIDS analysts are continuously forced to discard false positive (or less relevant true positives) in the course of their work. It is only through retrospective analysis in conjunction with an experienced NIDS analyst can we determine what information generated by a NIDS was the most critical information. Furthermore, while false positives are a significant issue with which a NIDS analyst must deal, false negatives can be an even greater risk to an institution – especially if the analyst does not have a way of knowing how many false negatives are occurring. Our work is a step in the direction of providing guidance to the analyst so that organizations which are constrained into using sampled data for NIDS purposes at least have a sense of how many anomalies their NIDS may be missing.

Chapter 17: Conclusions

This report unifies the work of two independent projects aimed at assisting resource-constrained network system analysts. A few overarching themes emerge:

- While particular constraints may make a problem theoretically impossible to solve, we can produce heuristic solutions that offer incremental improvements.
- When our constraints force us to utilize less than ideal circumstances – when, for example, we do not have the resources to monitor full network traces or we are forbidden by site policy to use NTP connections to maintain synchronization of distributed network sensors – we can at least use what is available to make estimations about what we may be missing.
- However, if we are unable to identify the “unknown unknowns”, we may falsely operate under naïve assumptions, and it is key that we identify and explore those assumptions to ensure that our estimations are correct.

It is our sincere hope that this work will contribute to the toolbox of network operators who may not have the opportunity or resources to carry out such examinations on their own local networks.

References

- [1] Braukhoff, D., Tellenbach, B., Wagner, A., May, M., and Lakhina, A. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the ACM Internet Measurement Conference* (October 2006), pp. 159-164.
- [2] Bro Intrusion Detection System. <http://www.bro-ids.org/>.
- [3] Bro Reference Manual. http://www.bro-ids.org/wiki/index.php/Reference_Manual.
- [4] Bro User Manual. http://www.bro-ids.org/wiki/index.php/User_Manual.
- [5] CAIDA data set OC48 Link A (San Jose, CA). Downloaded from <http://www.caida.org/data>, Feb 2007.
- [6] Duda, A., Harrus, G., Haddad, Y., and Bernard, G. Estimating Global Time in Distributed Systems. In *7th International Conference on Distributed Computing Systems*, Berlin, Germany, September 1987.
- [7] Duffield, N., Lund, C., and Thorup, M. Estimating flow distributions from sampled flow statistics. In *Proc. ACM/SIGCOMM*, 2003, pp. 325-336.
- [8] Firestorm. <http://sourceforge.net/projects/firestorm-ids>.
- [9] Ishibashi, K., Kawahara, R., Tatsuya, M., Kondoh, T., and Asano, S. Effect of sampling rate and monitoring granularity on anomaly detectability. In *Proc. 10th IEEE Global Internet Symposium*, 2007.
- [10] Jung, J., Paxson, V., Berger, A.W., and Balakrishnan, H. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy*, 2004, pp. 211-225.
- [11] Kohno, T., Broido, A., and Claffy, K.C. Remote physical device fingerprinting. In *IEEE Transactions on Dependable and Secure Computing*, 2005.
- [12] Kotz, D., Henderson, T., and Abyzov, I. CRAWDAD data set dartmouth/campus/tcpdump/fall0304 (v. 2004-11-09). Downloaded from <http://crawdad.cs.dartmouth.edu/dartmouth/campus/tcpdump/>, Feb 2007.
- [13] Mai, J., Chuah, C.-N., Sridharan, A., Ye, T., and Zang, H. Is sampled data sufficient for anomaly detection? *IMC 2006* (Rio de Janeiro, Brazil, October 2006).
- [14] Mai, J., Sridharan, A., Chuah, C.-N., Zang, H., and Ye, T. Impact of packet sampling on portscan detection. *IEEE Journal on Selected Areas in Communication* (2006).

- [15] Malmedal, Bjarte. Using netflows for slow portscan detection. Master's thesis, Department of Computer Science and Media Technology, Gjøvik University College, 2005.
- [16] Moon, S.B., Skelly, P., and Towsley, D. Estimation and removal of clock skew from network delay measurements. In *INFOCOM*, 1999.
- [17] Proebstel, E. and Hurd, S. "Characterizing and Improving Distributed Intrusion Detection Systems." Sandia Report, Sandia National Laboratories, December 2007.
- [18] Paxson, V. On calibrating measurements of packet transit times. In *SIGMETRICS*, 1998.
- [19] Ristenpart, T. Time stamp synchronization of distributed sensor logs: impossibility results and approximation algorithms. Master's thesis, Department of Computer Science, University of California, Davis, 2003.
- [20] SNORT. <http://snort.org>.
- [21] SNORT Reference Manual. Accessed at http://www.snort.org/docs/snort_htmanuals/htmanual_2.4/node11.html. Nov 2007.
- [22] Stewart, Aria. hdlc2en tool. <http://dinhe.net/~aredridel/projects/hdlc2en>.
- [23] Veitch, D., Babu, S., and Pasztor, A. Robust synchronization of software clocks across the Internet. In *IMC*, 2004.
- [24] Venables, W.N. and Ripley, B.D. *Modern Applied Statistics with S-Plus*. Springer Verlag, 1994.
- [25] Verzani, J. *Using R for Introductory Statistics*. CRC Press, 2004.

Appendix A. SNORT Results

This appendix provides the full results of SNORT sfportscan detection on the CAIDA and CRAWDAD datasets at various sampling rates. Where results for a particular sampling rate are not provided in the table, the number of scan alerts was zero; these table rows have been omitted for brevity.

A.1 SNORT low sense level

Table A.1 CAIDA Portsweeps at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	20	0	0.00%	20	0	0.00%	100.00%
1:5	1	0	0.00%	1	20	95.00%	5.00%

Table A.2 CRAWDAD Academic Hall Portsweeps at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	8221	0	0.00%	8221	0	0.00%	100.00%
1:5	1207	234	19.39%	973	7248	88.16%	11.84%
1:10	235	39	16.60%	196	8025	97.62%	2.38%
1:25	6	0	N/A	6	8215	99.93%	0.07%

Table A.3 CRAWDAD Residence Hall 100 Portsweeps at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	11882	0	0.00%	11882	0	0.00%	100.00%
1:5	194	28	14.43%	166	11854	98.60%	1.40%
1:10	21	1	4.76%	20	11881	99.83%	0.17%

Table A.4 CRAWDAD Residence Hall 13 Portsweeps at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
---------------	------------------	------------------	-----------------	----------------	-------------------	------------------	------------------

1:1	6507	6507	0.00%	6507	0	0.00%	100.00%
1:5	578	25	4.33%	553	6482	91.50%	8.50%
1:10	179	7	3.91%	172	6500	97.36%	2.64%
1:25	23	0	0.00%	23	6507	99.65%	0.35%

Table A.5 CAIDA Portscans at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	896	0	0.00%	896	0	0.00%	100.00%
1:5	190	147	77.37%	43	853	95.20%	4.80%
1:10	44	22	50.00%	22	874	97.54%	2.46%
1:25	19	4	21.05%	15	881	98.33%	1.67%
1:50	11	1	9.09%	10	886	98.88%	1.12%
1:75	1	0	N/A	1	895	99.89%	0.11%
1:100	1	0	N/A	1	895	99.89%	0.11%

Table A.6 CRAWDAD Academic Hall Portscans at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	105	0	0.00%	105	0	0.00%	100.00%
1:5	38	38	100.00%	0	67	100.00%	0.00%
1:10	4	3	75.00%	1	102	99.05%	0.95%

Table A.7 CRAWDAD Residence Hall 100 Portscans at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	403	0	0.00%	403	0	0.00%	100.00%
1:5	23	15	65.22%	8	388	98.01%	1.99%
1:10	1	1	100.00%	0	402	100.00%	0.00%

Table A.8 CRAWDAD Residence Hall 13 Portscans at low sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	263	0	0.00%	263	0	0.00%	100.00%
1:5	8	4	50.00%	4	259	98.48%	1.52%

A.2 SNORT medium sense level

Table A.9 CAIDA Portsweeps at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	57641	0	0.00%	57641	0	0.00%	100.00%
1:5	7697	254	3.30%	7443	50198	87.09%	12.91%
1:10	2246	101	4.50%	2145	55496	96.28%	3.72%
1:25	559	20	3.58%	539	57102	99.06%	0.94%
1:50	261	12	4.60%	249	57392	99.57%	0.43%
1:75	168	4	2.38%	164	57477	99.72%	0.28%
1:100	112	3	2.68%	109	57532	99.81%	0.19%
1:200	34	1	2.94%	33	57608	99.94%	0.06%

Table A.10 CRAWDAD Academic Hall Portsweeps at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	35911	0	0.00%	35911	0	0.00%	100.00%
1:5	32621	2965	8.26%	29656	6255	17.42%	82.58%
1:10	27533	367	1.33%	27166	8745	24.35%	75.65%
1:25	25624	38	0.15%	25586	10325	28.75%	71.25%
1:50	19117	11	0.06%	19106	16805	46.80%	53.20%
1:75	10750	4	0.04%	10746	25165	70.08%	29.92%
1:100	3363	6	0.18%	3357	32554	90.65%	9.35%
1:200	4	1	25.00%	3	35908	99.99%	0.01%
1:500	1	1	100.00%	0	35911	100.00%	0.00%

Table A.11 CRAWDAD Residence Hall 100 Portsweeps at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	13930	0	0.00%	13930	0	0.00%	100.00%
1:5	1007	396	39.32%	611	13534	95.61%	4.39%
1:10	188	42	22.34%	146	13888	98.95%	1.05%
1:25	4	1	25.00%	3	13929	99.98%	0.02%
1:50	1	0	0.00%	1	13930	99.99%	0.01%

Table A.12 CRAWDAD Residence Hall 13 Portsweeps at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	13930	0	0.00%	13930	0	0.00%	100.00%
1:5	1007	396	39.32%	611	13534	95.61%	4.39%
1:10	188	42	22.34%	146	13888	98.95%	1.05%
1:25	4	1	25.00%	3	13929	99.98%	0.02%
1:50	1	0	0.00%	1	13930	99.99%	0.01%

	alerts	scans			scans	rate	
1:1	17958	0	0.00%	17958	0	0.00%	100.00%
1:5	1381	392	28.39%	989	35519	94.49%	5.51%
1:10	593	67	11.30%	526	35844	97.07%	2.93%
1:25	356	1	0.28%	355	35910	98.02%	1.98%
1:50	78	0	0.00%	78	35911	99.57%	0.43%
1:75	47	0	0.00%	47	35911	99.74%	0.26%
1:100	31	0	0.00%	31	35911	99.83%	0.17%
1:200	2	0	0.00%	2	35911	99.99%	0.01%

Table A.13 CAIDA Portscans at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	5286	0	0.00%	5286	0	0.00%	100.00%
1:5	980	6	0.61%	974	4312	81.57%	18.43%
1:10	455	2	0.44%	453	4833	91.43%	8.57%
1:25	151	1	0.66%	150	5136	97.16%	2.84%
1:50	1	0	0.00%	1	5285	99.98%	0.02%

Table A.14 CRAWDAD Academic Hall Portscans at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	6	0	0.00%	6	0	0.00%	100.00%
1:5	62	61	98.39%	1	5	83.33%	16.67%
1:10	1		0.00%	1	5	83.33%	16.67%

Table A.15 CRAWDAD Residence Hall 100 Portscans at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	72	0	0.00%	72	0	0.00%	100.00%
1:5	9	8	88.89%	1	71	98.61%	1.39%
1:10	1	0	0.00%	1	71	98.61%	1.39%

Table A.16 CRAWDAD Residence Hall 13 Portscans at medium sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	39	0	0.00%	39	0	0.00%	100.00%
1:5	24	24	100.00%	0	39	100.00%	0.00%
1:10	2	2	100.00%	0	39	100.00%	0.00%

A.3 SNORT high sense level

Table A.17 CAIDA Portsweeps at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	57898	0	0.00%	57898	0	0.00%	100.00%
1:5	10561	311	0.00%	10250	47648	82.30%	17.70%
1:10	4040	112	2.77%	3928	53970	93.22%	6.78%
1:25	1011	47	4.65%	964	56934	98.34%	1.66%
1:50	353	11	3.12%	342	57556	99.41%	0.59%
1:75	206	6	2.91%	200	57698	99.65%	0.35%
1:100	154	6	3.90%	148	57750	99.74%	0.26%
1:200	61	0	0.00%	61	57837	99.89%	0.11%
1:500	25	0	0.00%	25	57873	99.96%	0.04%
1:1000	2	0	0.00%	2	57896	100.00%	0.00%

Table A.18 CRAWDAD Academic Hall Portsweeps at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	8221	0	0.00%	8221	0	0.00%	100.00%
1:5	1207	234	19.39%	973	7248	88.16%	11.84%
1:10	235	39	16.60%	196	8025	97.62%	2.38%
1:25	6	0	N/A	6	8215	99.93%	0.07%

Table A.19 CRAWDAD Residence Hall 100 Portsweeps at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	26565	0	0.00%	26565	0	0.00%	100.00%
1:5	2056	321	15.61%	1735	24830	92.34%	7.66%
1:10	612	106	17.32%	506	26059	97.77%	2.23%
1:25	82	18	21.95%	64	26501	99.72%	0.28%
1:50	10	2	20.00%	8	26557	99.96%	0.04%
1:75	6	2	33.33%	4	26561	99.98%	0.02%
1:100	3	2	66.67%	1	26564	100.00%	0.00%

Table A.20 CRAWDAD Residence Hall 13 Portsweeps at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	22649	0	0.00%	22649	0	0.00%	100.00%

1:5	1998	299	14.96%	1699	20950	92.50%	7.50%
1:10	750	90	12.00%	660	21989	97.09%	2.91%
1:25	189	17	8.99%	172	22477	99.24%	0.76%
1:50	93	4	4.30%	89	22560	99.61%	0.39%
1:75	73	2	2.74%	71	22578	99.69%	0.31%
1:100	63	2	3.17%	61	22588	99.73%	0.27%
1:200	42	1	2.38%	41	22608	99.82%	0.18%
1:500	4	0	0.00%	4	22645	99.98%	0.02%

Table A.21 CAIDA Portscans at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	6234	0	0.00%	6234	0	0.00%	100.00%
1:5	1277	43	3.37%	1234	5000	80.21%	19.79%
1:10	568	7	1.23%	561	5673	91.00%	9.00%
1:25	223	0	0.00%	223	6011	96.42%	3.58%
1:50	82	0	0.00%	82	6152	98.68%	1.32%
1:75	54	0	0.00%	54	6180	99.13%	0.87%
1:100	38	0	0.00%	38	6196	99.39%	0.61%

Table A.22 CRAWDAD Academic Hall Portscans at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	115	0	0.00%	115	0	0.00%	100.00%
1:5	182	172	94.51%	10	105	91.30%	8.70%
1:10	66	62	93.94%	4	6230	96.52%	3.48%
1:25	2	1	50.00%	1	6233	99.13%	0.87%

Table A.23 CRAWDAD Residence Hall 100 Portscans at high sense level.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	539	0	0.00%	539	0	0.00%	100.00%
1:5	77	53	68.83%	24	515	93.48%	6.52%
1:10	17	11	64.71%	6	533	98.37%	1.63%
1:25	2	2	100.00%	0	539	100.00%	0.00%

Appendix B. Bro Results

B.1 All Scans

Table B.1 All Scan alerts on CAIDA data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	266361	0	0.00%	266361	0	0.00%	100.00%
1:5	164665	45366	27.55%	119299	147062	55.21%	44.79%
1:10	97517	21791	22.35%	75726	190635	71.57%	28.43%
1:25	46821	6881	14.70%	39940	226421	85.01%	14.99%
1:50	26381	2528	9.58%	23853	242508	91.04%	8.96%
1:75	19186	1255	6.54%	17931	248430	93.27%	6.73%
1:100	15241	727	4.77%	14514	251847	94.55%	5.45%
1:200	7856	157	2.00%	7699	258662	97.11%	2.89%
1:500	2442	11	0.45%	2431	263930	99.09%	0.91%
1:1000	989	2	0.20%	987	265374	99.63%	0.37%

Table B.2 All Scan alerts on CRAWDAD Academic Hall data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	1:1	3148	0	0.00%	3148	0	0.00%
1:5	1:5	2730	18	0.66%	2712	436	13.85%
1:10	1:10	1700	20	1.18%	1680	1468	46.63%
1:25	1:25	1379	0	0.00%	1379	1769	56.19%
1:50	1:50	1277	12	0.94%	1265	1883	59.82%
1:75	1:75	1160	69	5.95%	1091	2057	65.34%
1:100	1:100	814	118	14.50%	696	2452	77.89%

Table B.3 All Scan alerts on CRAWDAD Residence Hall 13 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3629	0	0.00%	3629	0	0.00%	100.00%
1:5	2006	11	0.55%	1995	1634	45.03%	54.97%
1:10	1139	3	0.26%	1136	2493	68.70%	31.30%
1:25	458	1	0.22%	457	3172	87.41%	12.59%
1:50	233	0	0.00%	233	3396	93.58%	6.42%
1:75	151	0	0.00%	151	3478	95.84%	4.16%
1:100	95	0	0.00%	95	3534	97.38%	2.62%
1:200	19	0	0.00%	19	3610	99.48%	0.52%
1:500	3	0	0.00%	3	3626	99.92%	0.08%

Table B.4 All Scan alerts on CRAWDAD Residence Hall 100 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3589	0	0.00%	3589	0	0.00%	100.00%
1:5	2120	12	0.57%	2108	1481	41.26%	58.74%
1:10	1223	6	0.49%	1217	2372	66.09%	33.91%
1:25	488	1	0.20%	487	3102	86.43%	13.57%
1:50	228	0	0.00%	228	3361	93.65%	6.35%
1:75	171	0	0.00%	171	3418	95.24%	4.76%
1:100	101	0	0.00%	101	3488	97.19%	2.81%
1:200	29	0	0.00%	29	3560	99.19%	0.81%
1:500	1	0	0.00%	1	3588	99.97%	0.03%

B.2 Scan Analyzer

Table B.5 Scan analyzer alerts on CAIDA data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	24284	0	0.00%	24284	0	0.00%	100.00%
1:5	17426	2935	16.84%	14491	9793	40.33%	59.67%
1:10	12405	1675	13.50%	10730	13554	55.81%	44.19%
1:25	6489	771	11.88%	5718	18566	76.45%	23.55%
1:50	2310	232	10.04%	2078	22206	91.44%	8.56%
1:75	1335	113	8.46%	1222	23062	94.97%	5.03%
1:100	1028	51	4.96%	977	23307	95.98%	4.02%
1:200	545	3	0.55%	542	23742	97.77%	2.23%
1:500	145	0	0.00%	145	24139	99.40%	0.60%
1:1000	27	0	0.00%	27	24257	99.89%	0.11%

Table B.6 Scan analyzer alerts on CRAWDAD Academic Hall data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3068	0	0.00%	3068	0	0.00%	100.00%
1:5	2687	14	0.52%	2673	395	12.87%	87.13%
1:10	1682	18	1.07%	1664	1404	45.76%	54.24%
1:25	1373	0	0.00%	1373	1695	55.25%	44.75%
1:50	1274	12	0.94%	1262	1806	58.87%	41.13%
1:75	1160	69	5.95%	1091	1977	64.44%	35.56%
1:100	810	116	14.32%	694	2374	77.38%	22.62%

Table B.7 Scan analyzer alerts on CRAWDAD Residence Hall 13 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	572	0	0.00%	572	0	0.00%	100.00%
1:5	162	2	1.23%	160	412	72.03%	27.97%
1:10	74	0	0.00%	74	498	87.06%	12.94%
1:25	29	0	0.00%	29	543	94.93%	5.07%
1:50	9	0	0.00%	9	563	98.43%	1.57%
1:75	6	0	0.00%	6	566	98.95%	1.05%
1:100	5	0	0.00%	5	567	99.13%	0.87%
1:200	2	0	0.00%	2	570	99.65%	0.35%

Table B.8 Scan analyzer alerts on CRAWDAD Residence Hall 100 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	525	0	0.00%	525	0	0.00%	100.00%
1:5	146	1	0.68%	145	380	72.38%	27.62%
1:10	63	1	1.59%	62	463	88.19%	11.81%
1:25	5	0	0.00%	5	520	99.05%	0.95%

B.3 TRW-Scan Analyzer

Table B.9 TRW-Scan analyzer alerts on CAIDA data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	241542	0	0.00%	241542	0	0.00%	100.00%
1:5	147012	42423	28.86%	104589	136953	56.70%	43.30%
1:10	85017	20116	23.66%	64901	176641	73.13%	26.87%
1:25	40260	6110	15.18%	34150	207392	85.86%	14.14%
1:50	24070	2296	9.54%	21774	219768	90.99%	9.01%
1:75	17850	1142	6.40%	16708	224834	93.08%	6.92%
1:100	14212	676	4.76%	13536	228006	94.40%	5.60%
1:200	7311	154	2.11%	7157	234385	97.04%	2.96%
1:500	2297	11	0.48%	2286	239256	99.05%	0.95%
1:1000	962	2	0.21%	960	240582	99.60%	0.40%

Table B.10 TRW-Scan analyzer alerts on CRAWDAD Academic Hall data.

Sampling	# of	# of	False scan	Detected	# of	Missed	Consistency
----------	------	------	------------	----------	------	--------	-------------

rate	scan alerts	false scans	rate	scans	missed scans	scan rate	rate
1:1	28	0	0.00%	28	0	0.00%	100.00%
1:5	13	0	0.00%	13	15	53.57%	46.43%
1:10	12	1	8.33%	11	17	60.71%	39.29%
1:25	6	0	0.00%	6	22	78.57%	21.43%
1:50	3	0	0.00%	3	25	89.29%	10.71%

Table B.11 TRW-Scan analyzer alerts on CRAWDAD Residence Hall 13 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3047	0	0.00%	3047	0	0.00%	100.00%
1:5	1839	9	0.49%	1830	1217	39.94%	60.06%
1:10	1062	3	0.28%	1059	1988	65.24%	34.76%
1:25	429	1	0.23%	428	2619	85.95%	14.05%
1:50	224	0	0.00%	224	2823	92.65%	7.35%
1:75	145	0	0.00%	145	2902	95.24%	4.76%
1:100	90	0	0.00%	90	2957	97.05%	2.95%
1:200	17	0	0.00%	17	3030	99.44%	0.56%
1:500	3	0	0.00%	3	3044	99.90%	0.10%

Table B.12 TRW-Scan analyzer alerts on CRAWDAD Residence Hall 100 data.

Sampling rate	# of scan alerts	# of false scans	False scan rate	Detected scans	# of missed scans	Missed scan rate	Consistency rate
1:1	3054	0	0.00%	3054	0	0.00%	100.00%
1:5	1969	11	0.56%	1958	1096	35.89%	64.11%
1:10	1155	5	0.43%	1150	1904	62.34%	37.66%
1:25	483	1	0.21%	482	2572	84.22%	15.78%
1:50	228	0	0.00%	228	2826	92.53%	7.47%
1:75	171	0	0.00%	171	2883	94.40%	5.60%
1:100	101	0	0.00%	101	2953	96.69%	3.31%
1:200	29	0	0.00%	29	3025	99.05%	0.95%
1:500	1	0	0.00%	1	3053	99.97%	0.03%

Appendix C: NIDS Sampling Glossary

<i>Consistency Rate</i>	We define this term as the percentage of <i>real scans</i> that are detected by the IDS on a given sampled trace. It is calculated by dividing the number of <i>detected scans</i> by the number of <i>real scans</i> for the particular trace. It is also inversely proportionate to the <i>missed scan rate</i> .
<i>Detected Scan</i>	We define this as a <i>real scan</i> for which an alert is raised by an IDS on a particular sampled trace.
False Negative	An instance of anomalous traffic for which an IDS fails to raise an alert.
False Positive	An alert raised by an IDS that does not relate to anomalous traffic but, rather, is a result of benign traffic.
<i>False Scan</i>	We define this as an alert raised by an IDS on a sampled trace that does not correlate to a <i>real scan</i> .
<i>False Scan Rate</i>	We define this term as the percentage of alerts raised by an IDS on a sampled trace that are <i>false scans</i> . It is calculated by dividing the number of <i>false scans</i> by the number of <i>real scans</i> for the particular trace.
<i>Missed Scan</i>	We define this as a <i>real scan</i> for which no alert is raised by an IDS on a particular sampled trace.
<i>Missed Scan Rate</i>	We define this term as the percentage of <i>real scans</i> that are missed by the IDS on a given sampled trace. It is calculated by dividing the number of <i>missed scans</i> by the number of <i>real scans</i> for the particular trace. It is also inversely proportionate to the <i>consistency rate</i> .
<i>Real Scan</i>	We define this as a scan that is observed by the IDS in a full network trace.

Appendix D. Code for determining SNORT consistency

```
#!/usr/bin/perl
#
# Elliot Proebstel
# Fall 2007
#
# This code looks for false positives in sampled alert data.
# It compares alerts raised on sampled data to alerts
# raised on the full trace.
# For an alert from the sampled data to "match" an alert
# from the full trace:
#
# Using sense_level "low", the alerts must match on:
# * Source IP address
# * Alert type
# * Time window (60 seconds)
#
# Using sense_level "medium", the alerts must match on:
# * Relevant IP address (sourceIP for all but "decoy" or
# "distributed" scans)
# * Alert fields:
#   - (ICMP|TCP|UDP)
#     &&
#   - (Portscan|Portswep|Sweep)
# * Time window (90 seconds)
#
# Using sense_level "high", the alerts must match on:
# * Relevant IP address (sourceIP for all but "decoy" or
# "distributed" scans)
# * Alert fields:
#   - (ICMP|TCP|UDP)
#     &&
#   - (Portscan|Portswep|Sweep)
# * Time window (600 seconds)
#
# The code tracks matches (a maximum of one match per sample-data
# alert), # which it reports in a file that is cleverly named "matches",
# and false
# positives (alerts raised on sampled data that have no match in the
# full
# trace), which it reports in a file named "false_pos".
#
# The code requires a file called "all_portscans" to exist in the
# directory from which the script is run as well as in a
# directory accessible as ../full-trace
#
open(INPUT1,"<all_portscans"); # open "all_portscans" which has all
                              # portscan alerts
open(INPUT2,"<../full-trace/all_portscans"); # open "all_portscans"
                                              # in full trace
open(OUTPUT1,">matches");
open(OUTPUT2,">false_pos");

@alerts=<INPUT1>;
```

```

@baselines=<INPUT2>;

$totalMatches=0; # tracks total number of matches
$totalFPs=0;     # tracks total number of false positives

foreach $alert(@alerts) {

    $match=0;          # tracks current alert - matched yet or not

    @ids=split(/\}/, $alert);
    ($sourceIP, $destIP) = $ids[1] =~
m/(\d+\.\d+\.\d+\.\d+)\s+\-\>\s+(\d+\.\d+\.\d+\.\d+)/;
    $ids[0] =~ m/(\d\|)(.*)\(\|)/;
    $alertName=$2;
    if ($alertName =~ /(Distributed|Decoy)/) {$matchIP = $destIP;}
    else {$matchIP = $sourceIP;}
    $alertName =~ s/(\S+)/; # remove things in ( )
    $alertName =~ s/\s+//; # remove whitespace at the start
    ($alertProto, $alertType) = $alertName =~
m/(TCP|UDP|ICMP).+(PortswEEP|Portscan|Sweep)/;
    ($aHours, $aMins, $aSecs) = $ids[0] =~ m/(\d+):(\d+):(\d+)/;
    $aTotalSecs=(( $aHours*60*60)+( $aMins*60)+$aSecs);

    foreach $base(@baselines) {

        if ($match==0)
        {
            ($bHours, $bMins, $bSecs) = $base =~ m/(\d+):(\d+):(\d+)/;
            $bTotalSecs=(( $bHours*60*60)+( $bMins*60)+$bSecs);
            $timeDiff=abs($bTotalSecs-$aTotalSecs);

            if (($base =~ /$matchIP/) &&
($base =~ /($alertProto).+($alertType)/) && ($timeDiff <= 600))
            {
                print OUTPUT1 "$alert $base\n";
                $match=1;
                $totalMatches++;
            }
        }
    }
    if ($match==0)
    {
        print OUTPUT2 "no match for $alert";
        $totalFPs++;
    }
}
print OUTPUT1 "Total number of matches: $totalMatches\n";
print OUTPUT2 "Total number of false positives: $totalFPs\n";

close(INPUT1);
close(INPUT2);
close(OUTPUT1);
close(OUTPUT2);

```