Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic

Stefan Kornexl, *TU München*
Vern Paxson, *ICSI / LBNL*
Holger Dreger, *TU München*
Anja Feldmann, *TU München*
Robin Sommer, *TU München*

Abstract

There are times when it would be extraordinarily convenient to record the entire contents of a high-volume network traffic stream, in order to later "travel back in time" and inspect activity that has only become interesting in retrospect. Two examples are security forensics—determining just how an attacker compromised a given machine—and network trouble-shooting, such as inspecting the precursors to a fault after the fault. We describe the design and implementation of a *Time Machine* to efficiently support such recording and retrieval. The efficiency of our approach comes from leveraging the heavy-tailed nature of network traffic: because the bulk of the traffic in high-volume streams comes from just a few connections, by constructing a filter that records only the first N bytes of each connection we can greatly winnow down the recorded volume while still retaining both small connections in full, and the beginnings of large connections (which often suffices).

The system is designed for operation in Gbps environments, running on commodity hardware. It can hold a few minutes of a high volume stream in RAM, and many hours to days on disk; the user can flexibly configure its operation to suit the site's nature. We present simulation and operational results from three distinct Gbps production environments exploring the feasibility and efficiency of a Time Machine implementation. The system has already proved useful in enabling analysis of a break-in at one of the sites.

1 Introduction

Network packet traces—particularly those with not only headers but full contents—can prove invaluable both for trouble-shooting network problems and for investigating security incidents. Yet in many operational environments the sheer volume of the traffic makes it infeasible to capture the entire stream or retain even significant subsets for extended amounts of time. Of course, for both troubleshooting and security forensics, only a very small proportion of the traffic actually turns out to be pertinent. The problem is that one has to decide *beforehand*, when configuring a traffic monitor, what context will turn out to be relevant *retrospectively* to investigate incidents.

Only in low volume environments can one routinely bulk-record all network traffic using tools such as `tcpdump` [2]. Rising volumes inevitably require filtering. For example, at the *Lawrence Berkeley National Laboratory* (LBNL), a medium-size Gbps environment, the network traffic averages 1.5 TB/day, right at the edge of what can be recorded using commodity hardware. The site has found it vital to record traffic for analyzing possible security events, but cannot retain the full volume. Instead, the operators resort to a `tcpdump` filter with *85 terms* describing the traffic to skip—omitting any recording of key services such as HTTP, FTP data, X11 and NFS, as well as skipping a number of specific high-volume hosts, and all non-TCP traffic. This filter reduces the volume of recorded traffic to about 4% of the total.

At higher traffic rates, even such filtering becomes technically problematic. For example, the *Munich Scientific Research Network* (Münchner Wissenschaftsnetz, MWN), a heavily-loaded Gbps university environment, averages more than 2 TB external traffic each day, with busy-hour loads of 350 Mbps. At that level, it is very difficult to reliably capture the full traffic stream using a simple commodity deployment.

A final issue concerns *using* the captured data. In cases of possible security compromise, it can be of great importance to track down the attacker and assess the damage as quickly as possible. Yet, manually sifting through an immense archive of packet traces to extract a "needle in a haystack" is time-consuming and cumbersome.

In this work we develop a system that uses dynamic packet filtering and buffering to enable effective bulk-recording of large traffic streams. As this system allows us to conveniently "travel back in time", we term it a *Time Machine*. Our Time Machine buffers network streams first in memory and then on disk, providing several days of nearly-complete (from a forensics and trouble-shooting perspective) historic data and supporting timely access to locate the haystack needles. Our initial application of the Time Machine is as a forensic tool, to extract detailed past information about unusual activities once they are detected. Already the Time Machine has proved operationally useful, enabling diagnosis of a break-in that had gone overlooked at LBNL, whose standard bulk-recorder's static filter had missed capturing the relevant data.

Naturally, the Time Machine cannot buffer an entire high-volume stream. Rather, we exploit the "heavy-tailed" nature of network traffic to partition the stream more effectively (than a static filter can) into a small subset of high interest versus a large remainder of low interest. We then record the small subset and discard the rest. The key insight that makes this work is that most network connections are quite short, with only a small number of large connections (the heavy tail) accounting for the bulk of the total volume [6]. However, very often for forensics and trouble-shooting applications the *beginning* of a large connection contains the most significant information. Put another way, given a choice between recording some connections in their

entirety, at the cost of missing others in their entirety; versus recording the beginnings of all connections and the entire contents of most connections, we generally will prefer the latter.

The Time Machine does so using a *cutoff* limit, N: for every connection, it buffers up to the first N bytes of traffic. This greatly reduces the traffic we must buffer while retaining full context for small connections and the beginning for large connections. This simple mechanism is highly efficient: for example, at LBNL, with a cutoff of $N = 20$ KB and a disk storage budget of 90 GB, we can retain 3–5 days of *all* of the site's TCP connections, and, using another 30 GB, 4–6 days for all of its UDP flows (which tend to be less heavy-tailed).

We are not aware of any comparable system for traffic capture. While commercial bulk recorders are available (e.g, *McAfee Security Forensics* [3]), they appear to use brute-force bulk-recording, requiring huge amounts of disk space. Moreover, due to their black-box nature, evaluating their performance in a systematic fashion is difficult. Another approach, used by many network intrusion detection/prevention systems, is to record those packets that trigger alerts. Some of these systems buffer the start of every connection for a short time (seconds) and store them permanently if the session triggers an alert. Such systems do not provide long-term buffers or arbitrary access, so they do not support retrospective analysis of a problematic host's earlier activity. The Bro NIDS [5] can either record *all* analyzed packets, or *future* traffic once an incident has been detected. Finally, the Packet Vault system was designed to bulk record entire traffic streams [1]. It targets lower data rates and does not employ any filtering.

We organize the remainder of the paper as follows. In § 2, we briefly summarize the Time Machine's design goals. In § 3, we use trace-driven simulation to explore the feasibility of our approach for data-reduction in three high-volume environments. We discuss the Time Machine's architecture in § 4 and present an evaluation of its performance in two of the environments in § 5. § 6 summarizes our work.

2 Design Goals

We identified six major design goals for a Time Machine:

Provide raw packet data. The Time Machine should enable recording and retrieval of full packets, including payload, rather than condensed versions (e.g., summaries, or just byte streams without headers), in order to prevent losing crucial information.

Buffer traffic comprehensively. The Time Machine should manage its stored traffic for time-frames of *multiple days*, rather than seconds or minutes. It should not restrict capture to individual hosts or subnetworks, but keep as widespread data as possible.

Prioritize traffic. Inevitably, in high-volume environments we must discard some traffic quickly. Thus, the Time Machine needs to provide means by which the user can express different classes of traffic and the resources associated with each class.

Automated resource management. From experience, we know that having to manually manage the disk space associated with high-volume packet tracing becomes tedious and error-prone over time. The Time Machine needs to enable the user to express the resources available to it in high-level terms and then manage these resources automatically.

Efficient and flexible retrieval. The Time Machine must support timely queries for different subsets of the buffered data in a flexible and efficient manner. However, its packet capture operation needs to have priority over query processing.

Suitable for high-volume environments using commodity hardware. Even though we target large networks with heavily loaded Gbps networks, there is great benefit in a design that enables the Time Machine to run on off-the-shelf hardware, e.g., PCs with 2 GB RAM and 500 GB disk space.

3 Feasibility Study

In this section we explore the feasibility of achieving the design goals outlined above by leveraging the heavy-tailed nature of traffic to exclude most of the data in the high-volume streams.

Methodology: To evaluate the memory requirements of a Time Machine, we approximate it using a packet-buffer model. We base our evaluation on connection-level logs from the three environments described below. These logs capture the nature of their environment but with a relatively low volume compared to full packet-level data. Previous work [7] has shown that we can use flow data to approximate the data rate contributed by a flow, so we can assume that a connection spreads its total traffic across its duration evenly, which seems reasonable for most connections, especially large ones.

We evaluate the packet-buffer model in discrete time steps, enabling us to capture at any point the *volume* of packet data currently stored in the buffer and the *growth-rate* at which that volume is currently increasing. In our simplest simulation, the arrival of a new connection increases the growth-rate by the connection's overall rate (bytes transferred divided by duration); it is decreased by the same amount when it finishes. We then add the notion of keeping data for an extended period of time by introducing an *eviction time* parameter, T_e, which defines how long the buffer stores each connection's data. In accordance with our goals, we aim for a value of T_e on the order of days rather than minutes.

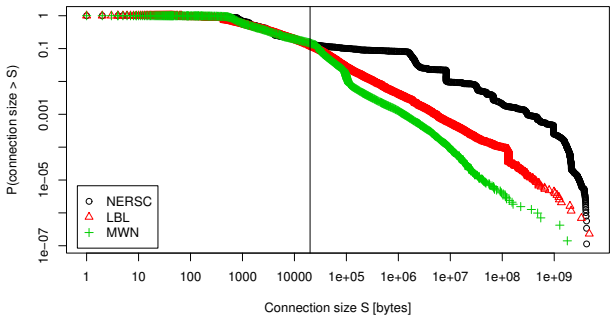

Figure 1: Log-log CCDF of connection sizes

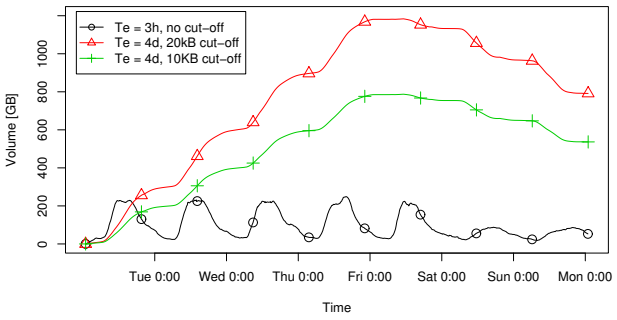

Figure 2: Simulated Volume for MWN environment

As described so far, the model captures bulk-recording with a timeout but without a *cutoff*. We incorporate the idea of recording only the first N bytes for each connection by adjusting the time at which we decrement the growth-rate due to each connection, no longer using the time at which the connection finishes, but rather the time when it exceeds N bytes (the connection *size cutoff*).

Environments: We drive our analysis using traces gathered from packet monitors deployed at the Internet access links of three institutions. While all institutions transfer large volumes of data (one to several TBs a day), their networks and traffic composition have qualitative differences.

MWN: The *Munich Scientific Research Network* (Münchner Wissenschaftsnetz, MWN) in Munich, Germany, connects two major universities and affiliated research institutions to the Internet, totaling approximately 50,000 hosts. The volume transferred over its Gbps Internet link is around 2 TB a day. Roughly 15–20% of the traffic comes from a popular FTP mirror hosted by one of the universities. The average utilization during busy-hours is about 350 Mbps (68 Kpps).

LBNL: The *Lawrence Berkeley National Laboratory* (LBNL) network in California, USA, comprises 9,000 hosts and 4,000 users, connecting to the Internet via a Gbps link with a busy-hour load of 320 Mbps (37 Kpps).

NERSC: The National Energy Research Scientific Computing Center is administratively part of LBNL, but physically separate and uses a different Internet access link; it provides computational resources (around 600 hosts) to 2,000 users. The traffic is dominated by large transfers, containing significantly fewer user-oriented applications such as the Web. The busy-hour utilization of the Gbps link is 260 Mbps (43 Kpps).

For our analysis we use connection-level logs of one week from MWN, LBNL, and NERSC. The MWN connection log contains 355 million connections from Monday, Oct. 18, 2004, through the following Sunday. The logs from LBNL and NERSC consist of 22 million and 4 million connections observed in the week after Monday Feb. 7, 2005 and Friday Apr. 29, 2005 respectively.

Analysis of connection size cutoff: As a first step we investigate the heavy-tailed nature of traffic from our environments. Figure 1 plots the (empirical) complementary cumulative distribution function (CCDF) of the number of bytes per connection for each of the three environments. Note that a "linear" relationship in such a log-log scaled plot indicates consistency of the tail with a Pareto distribution.

An important consideration when examining these plots is that the data we used—connection summaries produced by the Bro NIDS—are based on the difference in sequence numbers between a TCP connection's SYN and FIN packets. This introduces two forms of *bias*. First, for long-running connections, the NIDS may miss either the initial SYN or the final FIN, thus not reporting a size for the connection. Second, if the connection's size exceeds 4 GB, then the sequence number space will *wrap*; Bro will report only the bottom 32 bits of the size. Both of these biases will tend to *underestimate* the heavy-tailed nature of the traffic, and we know they are significant because the total traffic volume accounted for by the Bro reports is much lower than that surmised via random sampling of the traffic.

The plot already reveals insight about how efficiently a cutoff can serve in terms of reducing the volume of data the Time Machine must store. For a cutoff of 20 KB, corresponding to the vertical line in Figure 1, 12% (LBNL), 14% (NERSC) and 15% (MWN) of the connections have a larger total size. The percentage of bytes is much larger, though: 87% for MWN, 96% for LBNL, and 99.86% for NERSC. Accordingly, we can expect a huge benefit from using a cutoff.

Next, using the methodology described above we simulated the packet buffer models based on the full connection logs. Figures 2, 3 and 4 show the required memory for MWN, LBNL, and NERSC, respectively, for different combinations of eviction time T_e and cutoff. A deactivated cutoff corresponds to bulk-recording with a timeout. While the bulk-recording clearly shows the artifacts of time of day and day of week variations, using a cutoff reduces this effect, because we can accompany the cutoff with a much larger timeout, which spreads out the variations. We see that a cutoff of 20 KB quite effectively reduces the buffered volume: at LBNL, with $T_e = 4$ d, the maximum volume, 68 GB, is just a tad higher than the maximum volume, 64 GB, for bulk-recording with $T_e = 3$ h. However, we have increased the duration of data availabil-

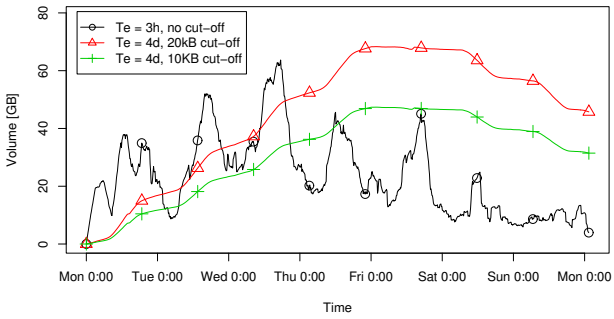

Figure 3: Simulated volume for LBNL environment

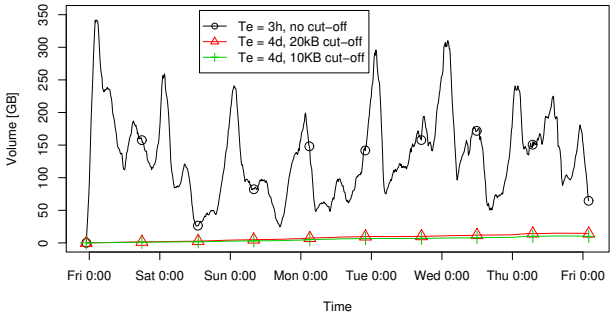

Figure 4: Simulated volume for NERSC environment

ity by a factor of 32! Note that the volume for simulations with $T_e = 4$ d stops to increase steadily after four days, since starting then connections are being evicted in the buffer model. At NERSC, the mean (peak) even decreases from 135 GB (344 GB) to 7.7 GB (14.9 GB). This enormous gain is due to the site's large proportion of high-volume data transfers. As already indicated by the lower fraction of bytes in the larger connections for MWN, the gain from the cutoff is not quite as large, likely due to the larger fraction of HTTP traffic.

Reducing the cutoff by a factor of two further reduces the maximum memory requirements, but only by a factor 1.44 for LBNL, 1.40 for NERSC, and 1.50 for MWN—not by a full factor of two. This is because at this point we are no longer able to further leverage a heavy tail.

The Figures also show that without a cutoff, the volume is spiky. In fact, at NERSC the volume required with $T_e = 1$ h is no more than two times that with $T_e = 1$ m, due to its intermittent bursts. On the other hand, with a cutoff we do not see any significant spikes in the volumes. This suggests that sudden changes in the buffer's growth-rate are caused by a few high-volume connections rather than shifts in the overall number of connections. All in all, the plots indicate that by using a cutoff of 10–20 KB, buffering *several days* of traffic is practical.

4 Architecture

The main functions our Time Machine needs to support are *(i)* buffering traffic using a cutoff, *(ii)* migrating (a subset of) the buffered packets to disk and managing the associated storage, *(iii)* providing flexible retrieval of subsets of the packets, and *(iv)* enabling customization. To do so, we use the multi-threaded architecture shown in Figure 5, which separates *user interaction* from *recording* to ensure that packet capture has higher priority than packet retrieval.

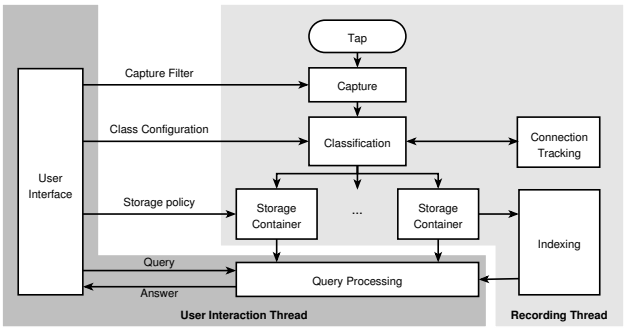

Figure 5: Time Machine System Architecture

The user interface allows the user to configure the recording parameters and issue queries to the *query processing* unit to retrieve subsets of the recorded packets. The recording thread is responsible for packet capture and storage. The architecture supports customization by splitting the overall storage into several *storage containers*, each of which is responsible for storing a subset of packets within the resources (memory and disk) allocated via the user interface. The *classification* unit decides which packets to assign to each storage container. In addition, the classification unit is responsible for monitoring the cutoff with the help of the *connection tracking* component, which keeps per connection statistics. To enable efficient retrieval, we use an index across all packets stored in all storage containers, managed by the *indexing* module. Finally, access to the packets coming in from the network *tap* is managed by the *capture* unit.

The capture unit receives packets from the network tap and passes them on to the classification unit. Using the connection tracking mechanism, it checks if the connection the packet belongs to has exceeded its cutoff value. If not, it finds the associated storage container, which then stores the packet in memory, indexing it in the process for quick access later on. It later migrates it to disk, and eventually deletes it. Accordingly, the actual Time Machine differs from the connection-level simulation model in that now the buffers are caches that evict *packets* when they are full, rather than evicting whole connections precisely at their eviction time.

Our implementation of the architecture uses the `libpcap` packet capture library [2], for which the user can specify a kernel-level BPF [4] capture filter to discard "uninteresting" traffic as early as possible. We collect and store each packet's full content and capture timestamp.

The capture unit passes the packet to the classification routines, which divide the incoming packet stream into classes according to a user-specified configuration. Each class definition includes a class name, a BPF filter to iden-

tify which packets belong to the class, a matching priority, and several storage parameters; for example:

```
class "telnet" { filter "tcp port 23";
  precedence 50; cutoff 10m;
  mem 10m; disk 10g; }
```

which defines a class "`telnet`" that matches, with priority 50, any traffic captured by the BPF filter "`tcp port 23`". A cutoff of 10 MB is applied, and an in-memory buffer of 10 MB and a disk budget of 10 GB allocated.

For every incoming packet, we look up the class associated with its connection in the connection tracking unit, or, if it is a new connection, match the packet against all of the filters. If more than one filter matches, we assign it to the class with the highest priority. If no filter matches, the packet is discarded.

To track connection cutoffs, the Time Machine keeps state for all active connections in a hash table. If a newly arrived packet belongs to a connection that has exceeded the cutoff limit configured for its class, it is discarded. We manage entries in the connection hash table using a user-configurable inactivity timeout; the timeout is shorter for connections that have not seen more than one packet, which keeps the table from growing too large during scans or denial of service attacks.

For every class, the Time Machine keeps an associated storage container to buffer the packets belonging to the class. Storage containers consist of two ring buffers. The first stores packets in a RAM buffer, while the second buffers packets on disk. The user can configure the size of both buffers on a per-class basis. (A key motivation for maintaining a RAM buffer in addition to disk storage is to enable near-real-time access to the more recent part of the Time Machine's archive.) Packets evicted from the RAM buffer are moved to the disk buffer. We structure the disk buffer as a set of files. Each such file can grow up to a configurable size (typically 10–100s of MB). Once a file reaches this size, we close it and create a new file. We store packets both in memory and on disk in `libpcap` format. This enables easy extraction of `libpcap` traces for later analysis.

To enable quick access to the packets, we maintain *multiple* indexes. The Time Machine is structured internally to support any number of indexes over an arbitrary set of (predefined) protocol header fields. For example, the Time Machine can be compiled to simultaneously support per-address, per-port, and per-connection-tuple indexes. Each index manages a list of time intervals for every unique key value, as observed in the protocol header field (or fields) of the packets. These time intervals provide information on whether packets with that key value are available in a given storage container and at what starting timestamp, enabling fast retrieval of packets. Every time the Time Machine stores a new packet it updates each associated index. If the packet's key—a header field or combination of fields—is not yet in the index, we create a new entry containing a zero-length time interval starting with the timestamp of the packet. If an entry exists, we update it by either extending the time interval up to the timestamp of the current packet, or by starting a new time interval, if the time difference between the last entry in the existing interval and the new timestamp exceeds a user-defined parameter. Thus, this parameter trades off the size of the index (in terms of number of intervals we maintain) for how precisely a given index entry localizes the packets of interest within a given storage container. As interval entries age, we migrate them from in-memory index structures to index files on disk, doing so at the same time the corresponding packets in the storage containers migrate from RAM to disk. In addition, the user can set an upper limit for the size of the in-memory index data structure.

The final part of the architecture concerns how to find packets of interest in the potentially immense archive. While this can be done using brute force (e.g., running `tcpdump` over all of the on-disk files), doing so can take a great deal of time, and also have a deleterious effect on Time Machine performance due to contention for the disk. We address this issue using the query-processing unit, which provides a flexible language to express queries for subsets of the packets. Each query consists of a logical combination of time ranges, keys, and an optional BPF filter. The query processor first looks up the appropriate time intervals for the specified key values in the indexing structures, trimming these to the time range of the query. The logical *or* of two keys is realized as the union of the set of intervals for the two keys, and an *and* by the intersection. The resulting time intervals correspond to the time ranges in which the queried packets originally arrived. We then locate the time intervals in the storage containers using binary search. Since the indexes are based on time intervals, these only limit the amount of data that has to be scanned, rather then providing exact matches; yet this narrowing suffices to greatly reduce the search space, and by foregoing exact matches we can keep the indexes much smaller. Accordingly, the last step consists of scanning all packets in the identified time ranges and checking if they match the key, as well as an additional BPF filter if supplied with the query, writing the results to a `tcpdump` trace file on disk.

5 Evaluation

To evaluate the Time Machine design, we ran an implementation at two of the sites discussed in § 3. For LBNL, we used three classes, each with a 20 KB cutoff: TCP traffic, with a space budget of 90 GB; UDP, with 30 GB; and Other, with 10 GB. To evaluate the "hindsight" capabilities, we determine the *retention*, i.e., the distance back in time to which we can travel at any particular moment, as illustrated in Figure 6. Note how retention increases after the Time Machine starts until the disk buffers have filled. After this

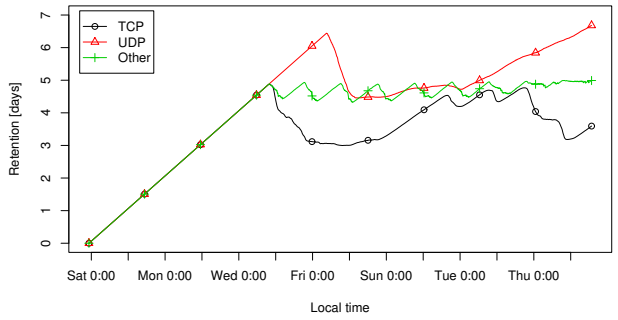

Figure 6: Retention in the LBNL environment

point, retention correlates with the incoming bandwidth for each class and its variations due to diurnal and weekly effects. New data forces the eviction of old data, as shown for example by the retention of TCP shortening as the lower level weekend traffic becomes evicted around Wed–Thu. The TCP buffer of 90 GB allows us to retain data for 3–5 days, roughly matching the predictions from the LBNL simulations (recall the volume biases of the connection-level data discussed in § 3). Use of a cutoff is highly efficient: on average, 98% of the traffic gets discarded, with the remainder imposing an average rate of 300 KB/s and a maximum rate of 2.6 MB/s on the storage system. Over the 2 weeks of operation, `libpcap` reported only 0.016% of all packets dropped.

Note that classes do not have to be configured to yield an identical retention time. The user may define classes based on their view of utility of having the matching traffic available in terms of cutoff and how long to keep it. For example we might have included a class configuration similar to the example in § 4 in order to keep more of Telnet connections for a longer period of time.

Operationally, the Time Machine has already enabled the diagnosis of a break-in at LBNL by having retained the response to an HTTP request that was only investigated three days later. The Time Machine's data both confirmed a successful compromise and provided additional forensic information in terms of the attacker's other activities. Without the Time Machine, this would not have been possible, as the site cannot afford to record its full HTTP traffic for any significant length of time.

At MWN we ran preliminary tests of the Time Machine, but we have not yet evaluated the retention capability systematically. First results show that about 85% of the traffic gets discarded, with resulting storage rates of 3.5 (13.9) MB/s average (maximum). It appears that the larger volume of HTTP traffic is the culprit for this difference compared to LBNL, due to its lesser heavy-tailed nature; this matches the results of the MWN connection-level simulation. For this environment it seems we will need to more aggressively exploit the classification and cutoff mechanisms to appropriately manage the large fraction of HTTP traffic.

The fractions of discarded traffic for both LBNL and MWN match our predictions well, and the resulting storage rates are reasonable for today's disk systems, as demonstrated in practice. The connection tracking and indexing mechanisms coped well with the characteristics of real Internet traffic. We have not yet evaluated the Time Machine at NERSC, but the simulations promise good results.

6 Summary

In this paper, we introduce the concept of a *Time Machine* for efficient network packet recording and retrieval. The Time Machine can buffer several days of raw high-volume traffic using commodity hardware. It provides an efficient query interface to retrieve the packets in a timely fashion, and automatically manages its available storage. The Time Machine relies on the simple but crucial observation that due to the "heavy-tailed" nature of network traffic, we can record most connections in their entirety, yet skip the bulk of the total volume, by storing up to (a customizable) cutoff limit of bytes per connection. We have demonstrated the effectiveness of the approach using a trace-driven simulation as well as operational experience with the actual implementation in two environments. A cutoff of 20 KB increases data availability from several hours to several days when compared to brute-force bulk recording.

In operational use, the Time Machine has already proved valuable by enabling diagnosis of a break-in that standard bulk-recording had missed. In future work, we intend to add a remote access interface to enable real-time queries for historic network data by components such as network intrusion detection systems.

7 Acknowledgments

This work was supported by the National Science Foundation under grant STI-0334088, and by a grant from the Bavaria California Technology Center, for which we are grateful.

References

[1] Antonelli, C., Undy, M., and Honeyman, P. The Packet Vault: Secure Storage of Network Data. In *Proc. Workshop on Intrusion Detection and Network Monitoring* (April 1999), pp. 103–110.

[2] Lawrence Berkeley National Laboratory. tcpdump and libpcap. `http://www.tcpdump.org/`.

[3] McAfee. McAfee Security Forensics. `http://www.mcafeesecurity.com/us/products/mcafee/forensics/security_for%ensics.htm`.

[4] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter 1993 Conference* (January 1993), pp. 259–270.

[5] Paxson, V. Bro: A system for detecting network intruders in real-time. *Computer Networks 31*, 23–24 (December 1999).

[6] Paxson, V., and Floyd, S. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking 3*, 3 (June 1995), 226–224.

[7] Wallerich, J., Dreger, H., Feldmann, A., Krishnamurthy, B., and Willinger, W. A Methodology for Studying Persistency Aspects of Internet Flows. *ACM SIGCOMM Computer Communication Review 35*, 2 (April 2005), 23–36.