Mnemosyne: Designing and Implementing Network Short-Term Memory

Giovanni Vigna Andrew Mitchell
Reliable Software Group
University California, Santa Barbara
{vigna,chemdog3}@cs.ucsb.edu

Abstract

Network traffic logs play an important role in incident analysis. With the increasing throughput of network links, maintaining a complete log of all network activity has become a task that requires an enormous amount of resources. We propose an approach to network monitoring that mitigates the resource consumption problem while still providing effective support to evidence collection and incident analysis. The approach relies on a tool, called MNEMOSYNE*, that maintains a sliding window containing the traffic that has been recently seen on a network link.* MNEMOSYNE *provides improved logging features, such as multiple streams, support for cross-stream queries, and dynamic remote reconfiguration. By integrating* MNEMOSYNE *with real-time intrusion detection capability, it is possible to provide incident analysis functionality and effective evidence collection, without having to maintain complete traffic logs. This paper describes the* MNEMOSYNE *tool, its architecture, and presents the results of the quantitative evaluation of its performance.*
Keywords: *Network Security, Intrusion Detection, Network Forensics, Incident Analysis*

1. Introduction

Incident analysis plays an important role in the everyday operation of computer networks. The goal of incident analysis is to understand what went wrong, gather evidence supporting the hypotheses, and, eventually, solve the problem that led to a security violation in the first place.

Incident analysis uses as input data the logs collected on affected hosts, routers, and network links. Collecting reliable and complete data about the activities that occur in a system is an issue by itself. Most operating systems offer some form of auditing that provides a log of the operations performed by different users. These logs can be limited to the security-relevant events in the system (e.g., failed login attempts) or they can be a complete report on every system call invoked by every process. For network activity, routers and firewalls also provide event logs. These logs can contain simple information, such as the opening and closing of network connections, or they can include a complete record of every packet that appeared on the wire.

A complete log of all traffic that has been transmitted over a network link is an invaluable resource during incident response. These logs represent the “memory” of the network, in terms of what happened and when. Once these logs are backed up and stored, they become the “long-term memory” of the network. Collecting and managing this information can be a daunting task, especially in the case of high-speed network links. Keeping up with a simple 100 Mbps link may require a substantial amount of resources and therefore most installations do not keep complete logs of network traffic. As a consequence, in most incident cases the relevant network traffic is lost forever and cannot be used as the basis for security analysis.

A possible alternative approach to complete network logging is to maintain a log that is limited to the network traffic that appeared recently on a network link. More precisely, the proposed approach relies on one or more packet capture applications that are responsible for collecting and maintaining one or more *sliding windows* of the traffic that has been sent on the network. A set of these applications can be seen as the “short-term memory” of the network.

Short-term memory information is available only for a limited amount of time. Therefore, this approach is particularly useful when combined with real-time intrusion detection and incident analysis. The basic idea is that when an attack is detected, e.g., by means of an intrusion detection system, a response application starts an incident analysis procedure. A first set of the traffic repositories is reconfigured to maintain a more persistent log of the traffic associated with the attack and, at the same time, the data stored so far is moved to permanent storage, so that it will not be lost. Preliminary analysis of the data contained in a repository may trigger queries to other repositories and may reconfigure the whole network memory infrastructure.

The packet capture applications supporting this approach cannot be simple logging applications. First of all, they must be able to maintain different traffic windows with different characteristics, such as window length, filters, and type of stored information. Second, they must be dynamically reconfigurable so that their behavior can be modified according to the current incident analysis process. Third, they must support queries on the collected data that are performed by applications during the incident response and analysis process.

Currently, there is no packet capture application that provides satisfactory support for the proposed incident analysis approach. Therefore, we have designed and developed a new packet capture application, called MNEMOSYNE, that effectively implements the concept of network "short-term memory". MNEMOSYNE is able to maintain multiple windows of network traffic with different characteristics, can be remotely reconfigured at run-time, and supports network traffic queries on the collected information.

This paper describes the design, implementation, and quantitative evaluation of the tool. More precisely, in Section 2 related work is surveyed. A reference model for packet capture applications is described in Section 3. Section 4 describes the design and implementation of MNEMOSYNE. Section 5 provides a quantitative evaluation of the tool, while Section 6 presents conclusions and outlines future work.

2. Packet Capture Applications

Packet capture is the act of acquiring network packets from a network interface. The interface is set in "promiscuous mode", allowing the application to capture all the traffic transferred over the local network link. Many applications support selective filtering of the traffic. This allows the user to select a specific subset of "interesting" packets, for example the traffic between two hosts, or the traffic involving a certain TCP port.

Different applications allow one to do different things with the received traffic. Common capabilities include pretty-printing of each packet to the console, saving all or a portion of each packet to a file, running intrusion detection software for malicious traffic, or generating usage statistics on the traffic in general.

An important aspect of packet capture applications is their ability to support queries. A query operates on the collected traffic and extracts a subset thereof that contains the relevant information. Queries are usually expressed in the same language that is used to specify filters.

One of the most well-known packet capture applications is *tcpdump* [4, 7] which is based on the *libpcap* packet capture library [3]. *tcpdump* reads packets from a network device and can be configured to use a selective filter specified using a filter expression language. *tcpdump* allows one to print the packets in a variety of formats to the console, or to save the packets to a file. The characteristics of the application are determined at invocation time and may not change during its execution. For example, it is not possible to ask an active *tcpdump* application to save a larger portion of the network packets or to save a different group of packets to a new file without stopping the current *tcpdump* instance and restarting a new one. In addition, *tcpdump* does not support queries natively. The common way to perform a query is to run another instance of *tcpdump* on a log file generated by a first instance. Using only *tcpdump*, queries across several log files are impossible to perform.

A natural improvement over one-stream packet capture is packet classification; that is the process of capturing multiple streams of traffic simultaneously.

One example of this type of application is *tcpflow* [8], which, like *tcpdump*, is based on *libpcap*. *tcpflow* operates on a single device and manages several streams. The application as a whole can be configured with a packet filter expressed in the *tcpdump* filter expression language. All packets are sorted by source and destination IP addresses, and then further classified by TCP port numbers and written to separate log files. The storage and selection characteristics (IP addresses and TCP ports) are predetermined, and may not be changed during execution. In addition, queries can not be performed without external help.

Another example of a packet classification application is *Ethereal* [1]. *Ethereal* operates on a single device and manages a single stream which can be associated with a global filtering expression. The filter associated with the input stream can be configured at execution time, and the capturing of the packets can be stopped and restarted during the execution of the application. There is a collection of pseudo-streams, called display streams, that allow the user to have subsets of the packets in the input stream presented in different colors. While a useful user interface feature, display streams are not saved to separate log files and cannot act as a query facility. *Ethereal* provides some basic support for capturing a sliding window of traffic. This feature, called "ring-buffer capture", uses a rotating queue of files that are used to store packets. The current graphic view of the stream is created from the files in the queue.

The product *EtherPeek NX* [10] is a prototypical example of a commercial packet classification application. *EtherPeek NX* operates on a single device and manages many streams that can be individually configured with filtering expressions expressed using a proprietary format. Each stream can be associated with an analysis module that performs operations such as generating an alarm, diagnosing common network problems, or storing traffic into a user-defined file. The storage characteristics of the traffic are determined at execution, but can be changed through the user interface. The filtering characteristics of existing streams can not be changed at run-time, though there is a mechanism to create additional streams at run-time. Simple queries are supported via the user-interface, but the application does not allow one to use an arbitrary filtering expression.

The program MNEMOSYNE is a packet capture application that overcomes the limitations of existing mainstream traffic dump applications. Even though MNEMOSYNE has been designed as a tool to support incident analysis, it can be used for a number of different tasks, including simple intrusion detection and statistical analysis of traffic. The most important characteristics of MNEMOSYNE are described below. The MNEMOSYNE tool is built on a library, called *libpclap*, that extends the functionality of the well-known *libpcap* library.

Higher configurability. MNEMOSYNE supports multiple streams and the filtering expression for each stream is a complete BPF expression. Streams can be organized and interconnected in arbitrary ways. The output of a stream can be used as input by other streams.

Higher flexibility. MNEMOSYNE storage parameters can be modified dynamically and remotely. MNEMOSYNE implements a control and configuration protocol that allows a system administrator to create new filters, change storage policies, etc.

Windowing. MNEMOSYNE in its typical configuration is able to maintain a sliding window of the captured traffic. The windowing parameters can be set dynamically according to time, size, or persistency.

Cross-stream queries. In addition to maintaining many streams at the same time, MNEMOSYNE allows one to perform queries across all the streams. This means that packets that appear in more than one stream are not duplicated in the query result.

3. A Model For Packet Capture Applications

This section introduces a reference model for packet capture applications. The reference model is used to describe in precise terms the characterizing components of packet capture applications. In addition, the model has been used in the design procedure and supports the evaluation and comparison of existing tools.

Packet capture applications retrieve packets from one or more sources. Examples are network interface cards (NICs), dial-up adapters, and also files generated by packet capture applications. In this model, any object that can be used as a source of network traffic is an *event source*. A packet capture application is characterized by a set of event sources $\phi = \{\phi_1, \phi_2, ..., \phi_n\}$.

An event source is represented as a sequence of meaningful occurrences, called *events*. Most events represent network packets. Although, a file-based event source may provide End-Of-File and Bad-Read events, and a kernel-based event source may provide a Kernel-Drop-Packet event. All events are identified by two integers, k and l that specify the event source that generated the event and the position of the event in the corresponding sequence. More precisely, the event source ϕ_k containing m events, is the ordered set $\{e_{k1}, e_{k2}, ..., e_{km}\}$. The set of all events, E, is thus defined by $E = \cup_i \phi_i$.

The entire sequence of events in an event source is not always needed. Typically, only a subset of the events generated by an event source, based on a certain selection criterion, is needed. For example, FTP traffic may be the only traffic of interest. A *filter* is the encapsulation of this selection process. Essentially, a filter decides the membership of an event to a specific group. As a membership function, a filter can be viewed as a collection of events that it will accept. We allow filters enough power to select any subset of E; in this way, the set of all filters F, is now $F = \wp(E)$. We define the function $accept : E \times F \mapsto \{true, false\}$, where $accept(e, f) = true \to e \in f$.

Packet capture applications select events through filters and perform some operations with the matching events. Common and useful actions include writing the event to a file, pretty-printing the event to the console, or executing a query. The actions to be undertaken are contained in a *callback*. There are many situations where system-specific information and the state of the application influence the action to be performed. The encapsulation of this information is called the *context*. We call the set of all callbacks C, and loosely define it to be the set of all functions that take an event and a context as input.

More formally, we define the context as the grouping of a history Q, an environment N, and a memory M. Also, we define the set W, which is the set of all possible contexts. The history Q is an ordered set containing all the events that have been accepted by a filter. The environment N is a collection of attribute-value pairs used to represent information about the context. The environment includes information such as network statistics and system properties. The memory M is a subset of Q containing the events that are available to the application. That is, the memory M contains the events currently stored in RAM, saved on disk, or that are otherwise accessible. Only the events in M are accessible by the callback via the context.

A *stream* is the coupling of a filter, a callback, and a context. The stream represents an autonomous processing unit. It contains all the relevant information to properly select events that are sent to it and handle them accordingly. The set of all streams is Σ. For a stream $S =< f, c, V >$, we define the functions $get\_filter : \Sigma \mapsto F$, $get\_callback : \Sigma \mapsto C$, and $get\_context : \Sigma \mapsto W$, where $get\_filter(S) = f$, $get\_callback(S) = c$, and $get\_context(S) = V$. Streams take events as inputs, filter them, and execute callbacks when a match is found. Matching events are propagated to the outputs. These events can be used as inputs by other streams.

A *packet capture application* is a system that interconnects event sources and streams in a meaningful way. We consider an application to be a directed graph G, which contains two types of vertexes. The first group of vertexes, V_e, represents the set of available event sources. The subgraph G_e formed out of the vertexes of V_e is an independent set, that is, there are no edges between vertexes in V_e. The second group of vertexes, V_s, represents the streams defined by the application. There are no restrictions on edges in the subgraph G_s, which is formed by the vertexes in V_s. No edges from G_s to G_e are allowed, though the edges from G_e to G_s can be arbitrary.

When an application is started, an event e is selected from an event source ϕ_i in G_e that has an event available. A copy of this event is sent on all outgoing edges from that vertex. When a stream vertex, v_{sj}, receives the event e, the accept function is called as $accept(e, get\_filter(v_{sj}))$. If the function maps to $false$, no additional processing occurs. Otherwise, the callback $c = get\_callback(v_{sj})$ is invoked with e and $w = get\_context(v_{sj})$ as parameters, that is $c(e, w)$. Then, a copy of the event is sent on all outgoing edges of v_{sj}. The application must guarantee that every edge maintains FIFO ordering, but no additional ordering at a higher level is required, though it may be provided.

There is more expressiveness and flexibility in this application model than what is required to describe most practical uses for packet capture applications. For example, a simple application like *tcpdump* contains only two vertexes. One is an event source, usually the default Ethernet NIC or a file, and the other is a stream, whose filter is specified as a command line parameter. The graph contains a single edge from the vertex v_{e1} to the vertex v_{s1}. The choice of callback is limited to display the event on the console or writing the event to a file.

Consider as another example the application *tcpflow*. Like *tcpdump*, *tcpflow* has only one event source v_{e1}. Unlike *tcpdump*, G_s is organized as a tree. The vertex v_{e1} is connected to the root of the tree in G_s. The tree satisfies the following property: at every level of the tree the traffic is partitioned. This means that a single event will take at most one accepting path from the root to a leaf. Every leaf stream has a callback that writes the event to a file.

4. MNEMOSYNE

MNEMOSYNE is designed to closely reflect the concepts introduced by the reference model. A MNEMOSYNE application is built by connecting streams and event sources. Reconfigurability is supported in several ways. First, the interconnections between streams and event sources can be changed at runtime in any way allowed for by the model. Second, MNEMOSYNE supports the creation and deletion of nodes in the graph G_s. That is, new streams can be added dynamically. Third, the filter, callback, or context of any stream can be changed at runtime.

The event sources contained in the MNEMOSYNE graph G_e are made available by the *event tap*. The event tap supports the creation of event sources from any physical device or capture file on the local machine. The current version does not support creation of event sources from files concurrently with physical devices. Either all event sources are files or they are all physical devices. The event tap component is provided by the *libpclap* library.

The *classifier* component is the manager and factory for the stream components. The stream component directly represents the vertexes of the graph G_s. The classifier provides for the creation of streams that contain a filter, a context, and a callback. The context does not contain the history Q, though the memory M has an implementation. The memory M is entirely contained in RAM. The usage of RAM can be configured by a variety of parameters. Furthermore, the memory behaves as a sliding window over the parameters of interest. Many statistics are available inside of the environment N. Some examples of the available statistics include number of packets, overall size of the packets stored in RAM, and timing information. The *libpclap* library provides the implementation of the classifier.

The memory M contained in a stream component is managed by the *storage unit*. The storage unit is a memory and file management system. Specifically, a storage unit is responsible managing the RAM that contexts may use to store network packets. A stream can select a set of parameters that determine how the memory associated with the stream's context is managed. Example parameters are RAM usage, number of packets, and age of the oldest packet. The storage unit uses a shared data space to store the network packets referenced in the memory of the streams. More precisely, the memory of a stream is actually a collection of references to packets inside a Flyweight pattern [1] This solution allows a set of streams that share a packet to require a single copy of the data. In addition to reducing the RAM requirement in many circumstances, the flyweight pattern yields a simple solution to eliminating duplicates from cross-stream queries.

Dynamic reconfiguration of a MNEMOSYNE instance is performed by means of control messages. Messages conform to the AMP protocol [5]. AMP messages can be used to create/destroy streams, change stream organization, modify stream parameters, and perform queries.

Consider, as an example, a MNEMOSYNE tool that has been started with an initial configuration that includes an AMP stream and a windowed stream L. The window contains two boundaries from 0 to 2 minutes of traffic and from 0 to 500 MB of traffic. Equivalently, this is the last 2 min-

[1] If instances of a class that contain the same information can be used interchangeably, the Flyweight pattern allows a program to avoid the expense of multiple instantiation by sharing a single instance [2].

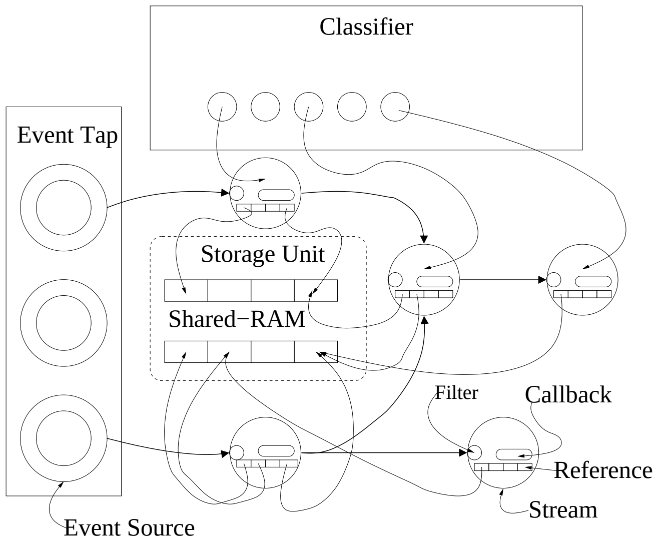

Figure 1. MNEMOSYNE **Components**

utes of traffic or the last 500 MB of traffic whatever is the less. Suppose that an intrusion is detected on host H on the same link. The real-time intrusion detection system realizes that the attack was perpetrated using a telnet connection from host E. Therefore, the intrusion detection system sends an AMP configuration message to the MNEMOSYNE active on the same link so that all the traffic regarding hosts E or H is now kept in a separate stream Z with a longer lifetime, say 2 hours. A configuration message is used to initialize the stream with a query that extracts the existing data from the original stream L. In addition, stream Z is marked as persistent so that no data will be lost.

AMP messages can be encapsulated in different, lower-level protocol data units (PDUs), such as ICMP messages, UDP datagrams, and TCP segments. AMP messages are delivered to a MNEMOSYNE application by injecting the messages into the traffic flow that is currently analyzed by the tool. The message processing is performed by a dedicated stream, called the AMP stream. The AMP stream filter selects the AMP messages out of the normal background traffic. The AMP stream callback is responsible for parsing the AMP message structure and perform the requested operations. By using the mechanism above to select and process control message it is possible to control remotely a MNEMOSYNE application even when it is executing on a host that does not have an associated IP address. In addition, by using the stream mechanism to select and process AMP messages it is possible to change the protocol used for message delivery at runtime.

5. Evaluation

To evaluate the performance of the approach, the MNEMOSYNE tool was installed on a network testbed. The configuration for the testbed includes two machines to perform the tests. One is required to run the software under evaluation and perform the measurements, while the other is responsible for generating the network traffic load.

The software to be tested was run on an Intel Pentium II processor clocked at 400MHz with 128MB RAM running RedHat Linux release 7.1 (Seawolf). The host has a 3COM 3C905B-TX Fast EtherLink XL card connected to the other machine through a hub.

The network traffic was generated on an Intel Pentium 4 processor clocked at 1.3GHz with 256MB RAM running RedHat Linux release 7.1 (Seawolf). The host has a builtin LAN port with the same driver as the other machine.

The generated traffic load contained simulated web traffic (23 percent of packet count and bandwidth), simulated user traffic (19 percent of packet count and bandwidth), simulated system traffic (48 percent of packet count and 47 percent of bandwidth), and simulated miscellaneous traffic (10 percent of packet count and 11 percent of bandwidth).

Traffic produced by the MIT Lincoln Laboratory for the DARPA 1998 Off-line Intrusion Detection Evaluation was used in a separate test.

The AMP protocol messages were injected into the network traffic using the MAC address of a “ghost” card, an unused IP address from the pool of addresses assigned to the testbed, and a specific pair of UDP ports. These features uniquely identified where the AMP packets were in the network traffic.

Given this configuration of the network, we decided to compare the performance of *tcpdump*, *Ethereal*, and

MNEMOSYNE. Each program was configured to simulate the behavior of *tcpdump*, the behavior of MNEMOSYNE with a 60 and 300 second window, and the behavior MNEMOSYNE with a 20 MB window. These tests will be detailed in the following sections.

5.1. MNEMOSYNE Simulating *tcpdump*

The typical usage of *tcpdump* is to run a single instance on a single machine capturing a single stream of traffic. The stream of traffic contains a filter that accepts all packets, and a callback that directly stores the packet into a file. Queries and analysis of the stream are performed by a second *tcpdump* instance, which is run after the first instance has completed capturing the traffic. One can control the maximum number of bytes that *tcpdump* obtains for a single packet. We used *tcpdump* version 3.6.2 invoked with a snapshot length of 2000 bytes.

Configuring MNEMOSYNE to reproduce the same functionality as *tcpdump* required several modifications. MNEMOSYNE was bootstrapped with a single stream that contained a filter and a callback. The filter accepts all packets. The callback stores packets to a file. The AMP stream was removed, which deactivated all of the dynamic configuration capabilities. The windowing functionality was disabled by configuring the storage unit to prevent packets from accumulating in RAM. The default maximum number of bytes of a captured packet in MNEMOSYNE is 65535, so no modification was necessary for that aspect of the capture process.

The test host was rebooted before each experiment. No program besides the performance analysis software and the program in question were executed during the test. A single iteration of the test consisted of the generator host transmitting both sets of traffic over the link between it and the software test host. After 10 iterations, the results were combined and averaged. The CPU usage from this test is presented in Figure 2. The RAM usage by both was negligible.

The CPU usage spike at the beginning of *tcpdump*'s execution is attributed to the accumulation of state associated with various traffic flows, such as relative sequence numbers for TCP streams. The accumulation and processing of state accounts for a significant portion of the difference in performance between MNEMOSYNE 's and *tcpdump*'s execution. This version of *tcpdump* did not provide a way to disable accumulation of state.

At 375 seconds into the test, the performance of *tcpdump* was slightly better than MNEMOSYNE. After an additional 15 seconds, the programs returned to their previous positions. For the majority of the test, MNEMOSYNE consumed half the CPU time as *tcpdump*.

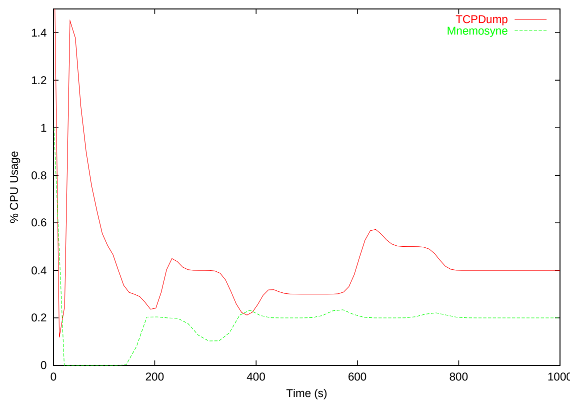

Figure 2. Performance evaluation of MNEMOSYNE **simulating** *tcpdump* **behavior.**

5.2. *Ethereal* Simulating *tcpdump*

The *Ethereal* program directly features the ability to simulate the relevant behavior of *tcpdump*. The GUI was used to begin a capture with the same properties as the previous test. The results of running *Ethereal* this way is shown with MNEMOSYNE in Figure 3.

The performance of both programs is very similar. *Ethereal* is less variant in usage during the first half of the test, while MNEMOSYNE is less variant during the second half of the test. Neither program utilized any significant portion of the CPU, nor consumed any significant amount of RAM.

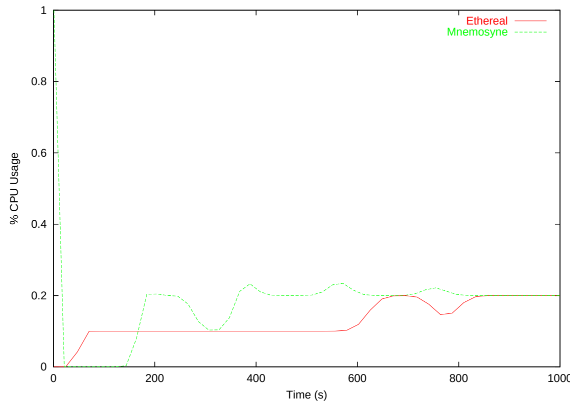

Figure 3. Performance evaluation of *Ethereal* **simulating** *tcpdump* **behavior**

5.3. *tcpdump* Simulating MNEMOSYNE

A typical MNEMOSYNE configuration is to run a single instance of the tool on a single machine, simultaneously capturing several streams of traffic. Each stream of traffic has a different filter, a different window, and a different callback. In this experiment, the chosen filters are the four filters that differentiate the four types of traffic from the first test set. Queries and analysis are performed across all streams.

There are five queries to be performed by this test. The first query, which was sent after 3 minutes, contained a filter that selects web traffic. The second query contained a filter that accepts all packets larger than 200 bytes. The second query was sent after 6 minutes. The third query contained a filter that accepts only traffic from a specific host, and was sent after 9 minutes. After 12 minutes, the fourth query was sent. The query contained a filter that accepts all traffic. The last query, whose filter rejects all packets, was sent after 15 minutes.

A single instance of *tcpdump* is incapable of providing a window of traffic. Providing windows of traffic was accomplished by running several overlapping instances of *tcpdump*. Two different methods of providing this window were examined. The first method provides a single file for assembling responses to queries at the expense of managing several processes and files. The second method attempts to provide a reduced program load at the expense of requiring additional processing assemble query responses from several files. The collection of *tcpdump* instances is incapable of providing multiple simultaneous streams of traffic. Providing multiple simultaneous streams requires several groups of *tcpdump* instances to be run in parallel with each other. There would need to be one group for each stream of traffic.

5.3.1 Fast Single File Query

The idea is to provide storage for traffic within a time window of size w with a granularity of size g. Once every g sized increment there would be a new complete file of size w that could be used for processing queries. To make this file available, the instance of *tcpdump* that created the file would have been started w size previous to the current moment, and would have been stopped at precisely the current moment. With $n = w/g$ instances of *tcpdump*, each equally spaced g apart, the entire window is covered by the available completed file. A query is implemented in this system by executing a separate *tcpdump* instance with the query's filter on that file.

A window size of $w = 60secs$ and $g = 1sec$ was chosen to begin the experiment. A script was used to coordinate the execution of the different *tcpdump* instances. Almost immediately, the software test host became unresponsive and dropped a significant portion of the packets. It also was unable to reliably run the performance evaluation software. The parameters for the script were changed to $g = 5secs$ to reduce the load placed on the host. In this configuration, it was possible to run the performance evaluation software without dropping packets.

Next, MNEMOSYNE was configured with $w = 60secs$ and $g = 1secs$. It is impossible and unnecessary to configure MNEMOSYNE with a granularity of $g = 5secs$. The $g = 1secs$ configuration provides a better time resolution, and could only run slower than the larger granularity.

Processing queries within MNEMOSYNE took on average $0.089secs$, and was bounded between $0.008secs$ and $0.172secs$. The *tcpdump* query processing took on average $5.054secs$, and was bounded between $4.094secs$ and $9.117secs$.

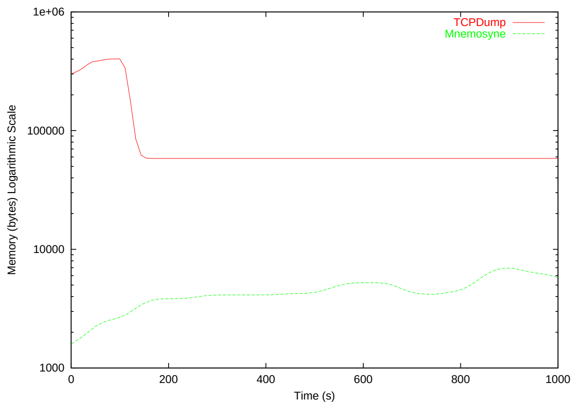

Figure 4. Memory usage of fast query *tcpdump* simulating MNEMOSYNE with 60 second window

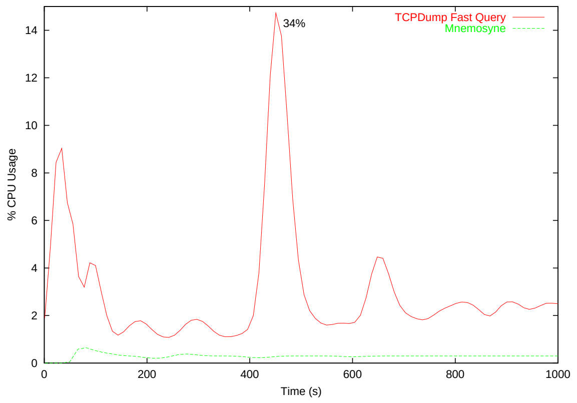

Figure 5. Performance evaluation of fast query *tcpdump* simulating MNEMOSYNE with 60 second window

The RAM usage of both MNEMOSYNE and the collection of *tcpdump* instances is depicted in the graph in Figure 4. This graph is in a logarithmic scale, and it is quite clear that MNEMOSYNE uses a few orders of magnitude less

RAM. The CPU usage is shown in the graph in Figure 5. MNEMOSYNE, after initialization, never rises above 1 percent CPU usage. The collection has a spike initially where half of the filters are accepting traffic. After $100secs$, the spike falls off, where it rises again at $400secs$. This is the location of the highest throughput of traffic. The spike caps at 34 percent of CPU usage, which is quite heavy in comparison to MNEMOSYNE's 0.2 percent usage in the same location.

For further comparison, the window size was expanded to $w = 300secs$ for MNEMOSYNE, and remained at $w = 60secs$ for *tcpdump*. The RAM usage of this instance of MNEMOSYNE and the collection of *tcpdump* instances is depicted in the graph in Figure 7. This graph is also in a logarithmic scale. It is evident that with this larger window MNEMOSYNE outperforms *tcpdump*, although the relative performance factor has been reduced from 100 to 2. The CPU usage is depicted in the graph in Figure 6. Besides an initial 3 percent usage in the first minute, MNEMOSYNE never uses more than 1 percent during the entire experiment.

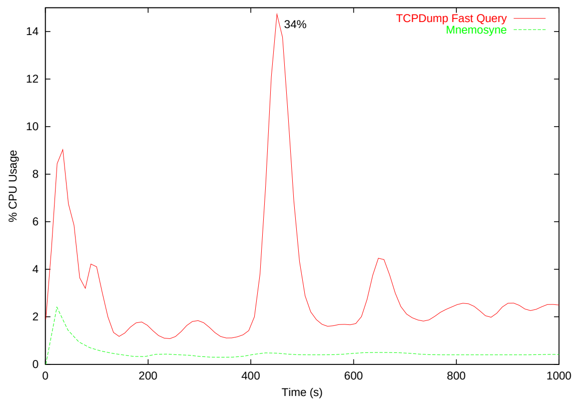

Figure 6. Performance evaluation of fast query *tcpdump* simulating MNEMOSYNE with 300 second window

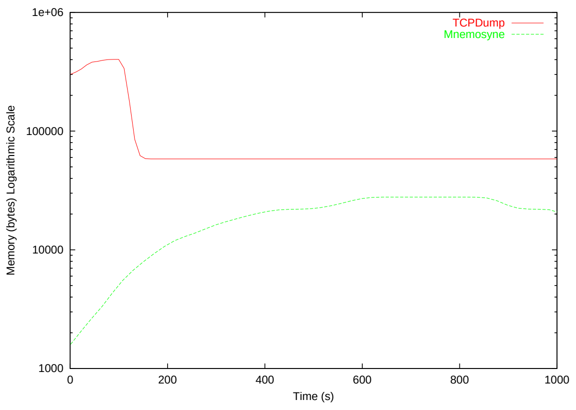

Figure 7. Memory usage of fast query *tcpdump* simulating MNEMOSYNE with 300 second window

5.3.2 Slow Multiple File Query

In this experiment the sliding window of size w with a granularity of size g is implemented in *tcpdump* using a smaller process load. Once every g sized increment there would be a collection of small files that could be accessed to reconstruct the entire window. To generate the files, three instances of *tcpdump* are started and killed in round-robin fashion. Every g size step, the next instance is killed and restarted. Each individual instance creates a sub-window of size $2g$ in a file. The file of each instance overlaps by g with the previous file and by g with the following file. A query is implemented by executing a program that finds the files that composes the entire window, and merges them together after applying the query's filter.

More specifically, a query is implemented as a merge-phase of an external merge-sort with special care taken to avoid duplicate packets. Each file contains a sorted collection of packets, but the files might not contain disjoint sets as in the case of adjacent sub-windows. When two files are being combined, the candidate packets are checked against each other for duplication, which causes the duplicate to be removed. When the final sorted duplicate-free file is formed, a linear walk is performed through the stored packets. A program was written to implement the merging and duplicate removal in optimal time $O(n)$.

To begin this phase of the experiment, a window size of $w = 60secs$ and $g = 1sec$ was chosen. The RAM usage is depicted in the graph in Figure 8. The RAM usage by the collection of *tcpdump* instances was essentially constant. After $500secs$, MNEMOSYNE briefly exceeds the memory usage of the *tcpdump* collection, only to exceed the usage again at $800secs$. The difference in memory usage is strongly offset by the file system usage by the *tcpdump* instances at $16megs$.

Processing the queries within MNEMOSYNE took an average of $0.089secs$, and was bounded between $0.008secs$ and $0.172secs$. *tcpdump* queries took an average of $11.917secs$, and was bounded between $4.094secs$ and $17.292secs$.

The CPU usage of both MNEMOSYNE and the collection of *tcpdump* instances is depicted in the graph in Figure 9. The collection performed more admirably this time settling on an average CPU usage of 2 percent. Even though this is 5 times as much usage as MNEMOSYNE, the more important difference is the variability of the usage. MNEMOSYNE rarely changes its CPU usage, where as the collection of *tcpdump* instances spikes occasionally to percentages around

20 and 30. During the spikes, the *tcpdump* would drop packets.

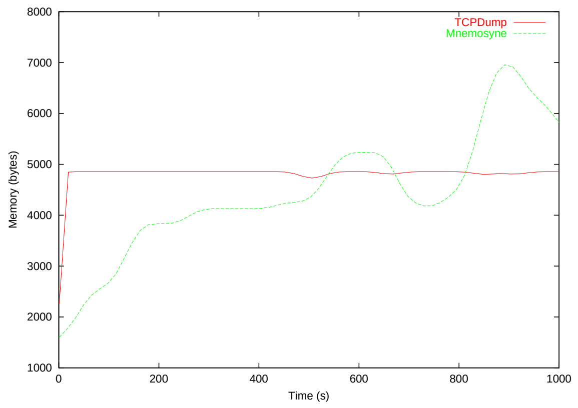

Figure 8. Memory usage of slow query *tcpdump* simulating MNEMOSYNE with 60 second window

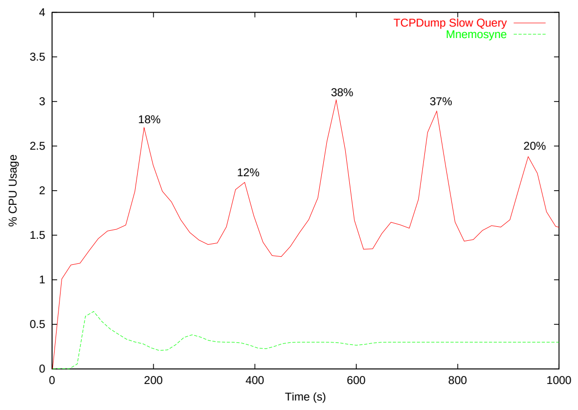

Figure 9. Performance evaluation of slow query *tcpdump* simulating MNEMOSYNE with 60 second window

To further this experiment, the window size was expanded to $w = 300secs$ for both programs. The RAM usage of this instance of MNEMOSYNE and the collection of *tcpdump* instances is depicted in the graph in Figure 11. With this larger window size, MNEMOSYNE uses almost 6 times more RAM than what was used by the collection of *tcpdump* instances. The collection of *tcpdump* instances uses $140megs$ of file system space, whereas MNEMOSYNE uses no file system space.

Processing the queries within the collection took an average of $14.316secs$, and was bounded between $5.548secs$ and $23.190secs$. MNEMOSYNE queries took an average of $0.091secs$, and was bounded between $0.011secs$ and $0.180secs$.

The CPU usage of both programs is shown in the graph in Figure 10. The collection of *tcpdump* instances has usages about 12 times that of MNEMOSYNE. The *tcpdump* instances are further burdened by the usage spikes that have percentages around 30 and 40. These spikes are the primary cause of the dropped packets in the collection program. MNEMOSYNE did not drop packets during execution, nor did the CPU usage ever rises to any significant percentage.

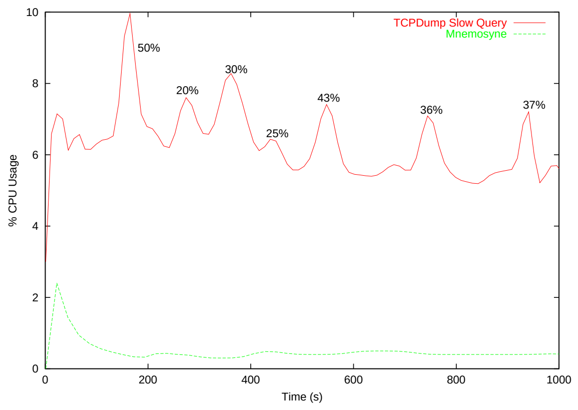

Figure 10. Performance evaluation of slow query *tcpdump* simulating MNEMOSYNE with 300 second window

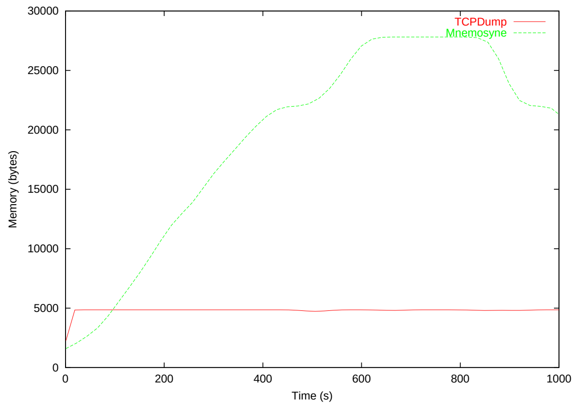

Figure 11. Memory usage of slow query *tcpdump* simulating MNEMOSYNE with 300 second window

5.4. *Ethereal* simulating MNEMOSYNE

Configuring *Ethereal* for this test was very similar to configuring *tcpdump*. *Ethereal* provides a rudimentary form of sliding window via a ring-buffer. The ring-buffer is a queue of files, where each file is filled to a specific number of bytes before cycling through to the next file. The contents of a previously held file are deleted at each step. *Ethereal* was configured to have 10 files in the ring-buffer, each of size 2MB. Essentially, *Ethereal* provided a window with the following properties: $w = 20MB$, and $g = 2MB$. Queries were handled in a manner identical to *tcpdump*.

Configuration for MNEMOSYNE differs from before in the following way: the window was changed to have the properties $w = 20MB$, and $g = 1B$.

The graph in Figure 12 depicts *Ethereal* with the query functionality disabled in the first test, and enabled in the second test. This is depicted with MNEMOSYNE running in the configuration described above. When the query functionality in *Ethereal* was disabled, the performances of MNEMOSYNE and *Ethereal* were close, with *Ethereal* performing slightly better. When the query functionality was enabled, the performance of MNEMOSYNE was consistently 3 times better than that of *Ethereal*. The time to process a query for *Ethereal* was identical to *tcpdump*.

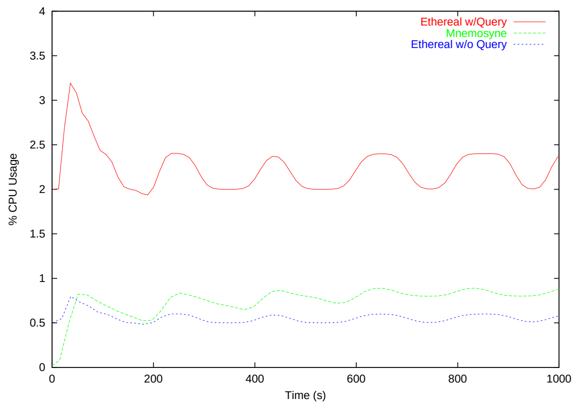

Figure 12. Performance evaluation of *Ethereal* **simulating** MNEMOSYNE **behavior**

6. Conclusions

MNEMOSYNE is a tool that maintains multiple streams of network traffic and allows one to re-configure type and parameter of the storage procedure. In addition, MNEMOSYNE supports cross-stream queries.

This paper described the design, implementation, and evaluation of the first prototype of the tool. Future work includes further testing, adding new functionality, and integration with other security tools.

A first added functionality will be the possibility for repositories to maintain a "degraded" version of the network traffic that drops outside a window (for example, by maintaining packet headers and throwing away payloads, or, by simply keeping track of the ports and addresses that have been used). The "aged" information is less complete but it is still a useful source of information that can serve as the basis for response and attacker tracking.

A second addition to the tool will be simple intrusion detection functionality aimed at mitigating the possibility of a denial of service attack against the repositories.

Integration of MNEMOSYNE with real-time intrusion detection is the next step in our research. We plan to integrate MNEMOSYNE repositories with real-time intrusion detection systems like NetSTAT [9] and Snort [6]. In addition, MNEMOSYNE will be integrated in an ongoing project at Dartmouth College for the creation of a storyboard system in support to incident handling. Withing the proposed system the security analysis is performed by ad hoc mobile programs that are injected in the network and move from repository to repository tracing back the "history" of a network attack. By using analysis tools based on mobile code it will be possible to achieve a high-level of configurability and customizability.

Acknowledgments

This research received partial support from ISTS, Dartmouth College, contract number 5-36285.

References

[1] The Ethereal Network Analyzer. `http://www.ethereal.com/`, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[3] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, January 1993.

[4] S. McCanne, C. Leres, and V. Jacobson. Tcpdump 3.4. Documentation, 1998.

[5] A. L. Mitchell. Mnemosyne - Short-Term Memory for Networks. Technical report, Dept. of Computer Science, UCSB, 2002.

[6] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.

[7] Tcpdump and Libpcap Documentation. http://www.tcpdump.org, Jan 2002.

[8] tcpflow - - TCP Flow Recorder. `http://www.circlemud.org/$\sim$jelson/software/tcpflow/`, June 2001.

[9] G. Vigna and R. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[10] Network Analysis Software, Training and Certification - WildPackets. `http://www.wildpackets.com/`, 2002.