Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

# Ruminate: A Scalable Architecture for Deep Network Analysis

**Charles Smutz** csmutz@gmu.edu
**Angelos Stavrou** astavrou@gmu.edu

Technical Report GMU-CS-TR-2010-20

## Abstract

Traditionally, Network Intrusion Detection Systems (NIDS) inspect packet header and payload data for malicious content. While each system is different, most NIDS perform limited analysis on network streams and network protocols. Unfortunately, current NIDS are typically susceptible to evasion through network protocol encoding, such as base64 encoding of SMTP/MIME or gzip compression of HTTP. In addition, malicious desktop application payloads (e.g., PDF documents, Flash multimedia files) are beyond the inspection capabilities of popular NIDS.

To address these limitations, we introduce Ruminate, a scalable object-centric traffic inspection and analysis architecture. Ruminate provides a distributed platform for deep analysis of network payload content. This includes full decoding of network protocols and recursive extraction of client application objects transferred over the network. While traditional NIDS utilize static packet load balancing to provide scalability, Ruminate employs dynamic load distribution of reassembled network streams and embedded objects, outsourcing the heavy processing to other processors or connected hosts. Therefore, high latency or computationally expensive analysis can be performed on commodity servers. Furthermore, our approach empowers system administrators to provision resources and preferentially treat traffic not only depending on the packet header but also on the data objects it carries. To achieve this, each object inspection algorithm is implemented as a separate component or service offered through a highly scalable producer-consumer architecture. We demonstrate using real-world traffic that our load balancing is far superior to existing techniques. This is because its granularity depends on the reconstructed objects rather than packet or simple stream analysis. Unlike existing systems, Ruminate can prevent NIDS evasion that leverages encoding or compression of malicious objects in network protocols, desktop application file formats, or encapsulation within other objects.

## 1 Introduction

Network-born attacks continue to move up the stack [29] from network layer attacks against remotely accessible system daemons to multiply encoded content attacks targeting desktop applications. Indeed, vulnerabilities in popular desktop applications, such as document readers (Adobe PDF and Microsoft Office), multimedia (Adobe Flash and Apple Quickime), and browsers (Internet Explorer and Firefox) are chief among the remote exploitation mechanisms. Some sources indicate that the number of vulnerabilities exploited in user applications exceeds the number of vulnerabilities in operating systems [13, 22]. In addition, targeted attacks often leverage a very high level of social engineering, further advancing these attacks up the stack from exploitation of technical vulnerabilities to exploitation of user behavior. The recognition of highly-persistent targeted attacks, including those attributed to the Advanced Persistent Threat (APT) [11], as an important emerging attack class provides a strong impetus for the capability to detect attacks that exploit client application vulnerabilities and social engineering. Maintaining a defense against persistent attackers also requires extensive network forensics capabilities allowing for correlation of related activities spanning large time frames.

Current Network Intrusion Detection Systems (NIDS) [26, 32, 24, 33] are focused on inspecting packets and, in some cases, offer pattern matching for popular protocols. However, they steer-off from extracting embedded objects from streams and detect application attacks as they traverse the network. Exposing attacks that hide inside application objects requires reordering, parsing, and decoding that involves many packets. Extracting the object from the network often requires recursive decoding and de-compression in network protocols, which obscure packet payloads from direct analysis. For instance, the base64 encoding used in MIME or the gzip compression used in HTTP render the extraction and inspection of network traffic a computationally hard problem. Another complication

stems from the fact that some of the extracted objects can often contain other embedded objects. For example, an attack that exploits a vulnerability in a multimedia renderer may be embedded in an otherwise innocuous container such as a document. The exploit is automatically triggered upon opening of the document by calling the multimedia renderer. To make matters worse, successfully analyzing packets as they traverse the network often requires that both network protocols and client application objects be parsed and decoded. This level of analysis is currently not supported in a scalable fashion by NIDS.

In this paper, we introduce Ruminate, an object-focused inspection and analysis architecture. Ruminate provides a platform for performing analysis of client application objects embedded within network traffic. The primary goal is complete decoding/decapsulation of network protocols and client objects. As such, recursive analysis is supported. Analyzers are built using a service-oriented approach wherein each analyzer is designed to accept a given data type, which can extract embedded objects for further analysis. For example, network packets are reassembled into TCP streams, which are then are processed by network protocol analysis services such as HTTP and SMTP/MIME. Payloads are analyzed by their respective parsing services (e.g., PDF document, ZIP archive). These services implement extensive parsing and transaction audit log generation, in addition to other possible detection mechanisms such as signature matching or dynamic analysis. Ruminate identifies any nested objects and submits them to the appropriate service for analysis. For example, the ZIP archive analysis service extracts compressed objects including JPEG images, PDF documents, or even other archives for recursive analysis. Ruminate seeks to ensure complete recursive analysis, if desired, in order to counter evasion through network or embedded-object encoding.

Ruminate can prevent NIDS evasion attacks that employ: a) encoding or compression of malicious objects in the network protocol (e.g. HTTP gzip) b) encoding or compression of embedded file formats (e.g. compression in PDF) and c) encapsulation within other objects (PDF inside a ZIP file). We do so by performing recursive decoding and unpacking of protocols and embedded objects. Though Ruminate does not introduce any new detection techniques, it provides a scalable and modular framework for network payload analysis. To measure the efficiency of our approach, we have implemented a prototype of Ruminate. Furthermore, we installed Ruminate on a small cluster of commodity machines and we were able process and analyze the network traffic generated by the users of an entire university campus without any packet loss. Our results show that we can efficiently use commodity hardware to fully decode and audit network protocols, and to inspect any and all objects that leave or enter an organization. We also demonstrate the ability of Ruminate to perform object inspection and detection on live network traffic. We demonstrate proficient detection of malicious PDF documents, which requires network protocol decoding, client application specific file format parsing, and computationally-expensive analysis.

Moreover we were able to demonstrate that our approach is capable of providing scalable detection of application-level malfeasance. For the empirical evaluations of Ruminate, we used datasets collected from the Internet router of a 30,000-student university campus, consisting of 973,718,083 packets and totaling 737 Gb. With this approach, we managed to load-balance 64 connected nodes and to detect malware that spanned the entire campus connection. Our goal was to show that a framework like Ruminate has become a necessary tool to complement the operation of current signature-based NIDS in ferreting out known threats.

In summary, our contributions are:

- A scalable object-centric network payload analysis architecture that is modular and enables the distributed deep inspection of network traffic on commodity hardware. This framework includes dynamic load-balancing of transport layer streams, full network protocol decoding, and a service oriented approach to recursive embedded object analysis.
- Ruminate acts as the conduit between different packet, flow, object decapsulation, and object analysis mechanisms enabling the extraction and analysis of deeply embedded data objects. Ruminate readily integrates with existing detection mechanisms, including those not designed specifically for NIDS.
- Ruminate empowers network and system operators to load-balance network traffic based on stream content rather than mere packet header and protocol information. Resources and detection mechanisms can be adjusted dynamically.

## 2 Related Work

The majority of recent NIDS research [33] focuses on improving the performance of NIDS through improving the performance of detection mechanisms such as signature matching [26, 32, 24]. For example, improvements over non-deterministic finite-state automata (NFA) are demonstrated by Yu et al. [38] through the use of deterministic finite-state automata (DFA) and by Smith et al. [30, 31] through the application of extended finite automata (XFA). Duncan and Jungck [15] extend the capabilities of special purpose hardware and the packetC language, allowing additional operations and data storage mechanisms to be implemented efficiently. While this research offers faster implementation of existing detection techniques, namely signature matching

and packet inspection, there is little focus on providing functionality necessary for embedded object analysis. Another thread of research has focused on string and regular expression matching for network intrusion detection [34, 36, 19, 21, 16]. All these approaches, while scalable, cannot offer robust protection against adversaries that target desktop applications and end users. Indeed, their matching mechanisms can be easily bypassed through commonplace encoding, compression, or other object embedding techniques. To make matters worse, in most cases, their filters can be easily miss attacks that extend beyond a single packet.

Other research focuses on improving the performance of NIDS on commodity hardware through improvements in scalability. Paxon et al. [27] and Vallentin et al. [35] demonstrate significant progress towards a NIDS cluster that can scale through the use of parallel NIDS analysis nodes but which still supports analysis across all nodes. Various load balancing schemes based on packet header information are explored including both network layer and transport layer schemes. This system, based on Bro running on commodity hardware, demonstrated the capability to scale beyond the capacity of a single NIDS instance using commodity hardware, with the exception of special purpose hardware for the load balancing.

Deri and Fusco [12] implemented an extension to their PF_RING Linux kernel modification to better support multithreaded or multiple process analysis. The new extension, called threaded network application programming interface (TNAPI), allows multiple processor cores to service independent packet queues simultaneously and efficiently provide network packets to multiple independent monitoring applications without the bottleneck of a single kernel socket. An open issue identified is that the effectiveness of this mechanism is reliant on even distribution of load by the packet header hashing mechanism. Gu et al. [17] presents a very short study on the issue of transport layer load balancing algorithms as applied to NIDS and demonstrates that alternative mechanisms for load balancing, including ones that require tracking connection state and/or knowledge of load on individual nodes, can provide more efficient load balancing than more naive approaches. Ruminate demonstrates the value of implementing a more granular load distribution, especially dynamic load balancing of transport layer sessions and embedded objects.

Furthermore, there is a plethora of mechanisms that attempt to automatically classify network packets and identify the corresponding protocol or attribute the communication to a specific application [23, 10, 18, 25, 8, 14]. In addition, there are techniques that attempt to identify the network traffic characteristics using visual motifs [20]. All of the above techniques can be used as a component in our modular architecture and feed protocol specific analysis engines that can further process the contents of the streams extracting objects and data for subsequent inspection. Such engines include [9, 28, 37] among others can be applied in parallel to streams of packets after a protocol has been assigned to this stream. In addition, VRT Razorback [6] advances an alternative implemention of a client application object analysis platform allowing the use of arbitrary engines, but does not present a mechanism for comprehensive and scable analysis of network protocols in order to extract all embedded objects from network traffic. Our aim is to make these systems operate under the same umbrella by becoming the "glue" that ties them together in a scalable and resource aware architecture.

# 3 System Architecture

Ruminate provides a scalable architecture for performing packet re-assembly and re-ordering, decapsulation, decompression, and analysis of desktop application objects nested within network traffic. Our primary goal was to achieve full decoding/decapsulation of network protocols through recursive analysis. To that end, we built a consumer-producer, service-oriented system consisting of individual analysis engines that can parse and extract different protocols, data types, and objects. Analysis engines are fitted in a service-oriented approach where each analyzer is designed to accept a given data type, perform analysis and alerting, and submit any embedded objects to the appropriate service. For example, network packets are reassembled into TCP streams, that are then processed by network protocol analysis services such as HTTP and SMTP/MIME. Packets and protocols are redirected and are processed by their respective parsing services at different levels. For instance, HTML stream, email, PDF document, a ZIP archive. Some of these services implement extensive transaction audit log generation in addition to other possible detection mechanisms such as signature matching or dynamic analysis. Since some objects may contain other objects, objects extracted from an analysis service are submitted to the appropriate service for further analysis. For instance, the ZIP archive analysis service can decompress objects, such as JPEG images, PDF documents, or even other archives for recursive analysis. Ruminate seeks to ensure complete recursive analysis, if desired, to counter evasion through network or embedded object encoding. While not required in real-world testing and usage, limits or prioritization could be used to mitigate the effects of resource exhaustion and "blinding" attacks.

One of the design tenets for Ruminate was the ability to operate on commodity servers, including cloud architectures. This constraint requires that Ruminate provide more efficient load distribution mechanisms than are currently used in NIDS, especially as the complexity and latency of the analysis performed increases. An architecture that can utilize existing analysis mechanisms with as few modifications as possible is also desirable. The

service-based approach accommodates this and allows for analysis mechanisms to be implemented in various programming languages or even on different platforms. The desire to leverage off-the-shelf hardware and software is a driving factor behind the architecture of Ruminate. This architecture also supports agility in detection capabilities and allows for services to be modified or added by changing the embedded-object analyzers that apply across all protocols.

To facilitate analysis on network payload data, including computationally-expensive or high-latency analysis, a more efficient load-balancing mechanism is required. The current NIDS load-balancing mechanisms operate on packets and distribute load to analysis pipelines that are largely independent. The efficiency of this load distribution method breaks down as the number of analysis pipelines increases, as the latency of analysis increases, or as the variance in the complexity of analysis increases. Conversely, Ruminate provides for dynamic load distribution at the network stream and object level.

Ruminate separates the processing of each object type into modules or services. For example, packets are reassembled into streams by one service, another service implements each network application protocol (e.g., HTTP, SMTP), and each client application payload (e.g., PDF documents, ZIP archives) are processed by a third service. The separation of these tasks into discrete services provides various advantages, including the ability to perform a more granular load distribution, as well as a flexibility in implementation and infrastructure. Since services operate on a single object type (and only on that type), we can take advantage of the locality of reference of data regarding the target object type. This can help to simplify the problem of inter-analyzer coordination, thereby facilitating cross-object analysis. After packets are reassembled, the resulting network streams are placed into queues according to protocol. Each queue can be prioritized relative to the others. Each stream that is ready to be processed is then passed to an available network application analyzer. Network analyzers subscribe to feeds of streams based on type and indicate to the stream reassembly component when they are available to processes additional streams. If a network analyzer processes a stream containing an embedded object, that object is extracted and made available to the appropriate object analyzer server. Each object analysis service parses objects, creates a transaction log that can be audited, extracts any embedded objects, and performs detections such as signature matching. The methods of load distribution for embedded object analyzers are not proscribed by Ruminate. The reference object analysis services presented here (ZIP archive and PDF document) both use a web service interface, which allows them to leverage web service load distribution mechanisms. While not explicitly demonstrated here, this architecture allows for different services to operate simultaneously on the same object, and it provides parallelism by separating the various layers of analysis.

Ruminate provides a platform that is capable of utilizing existing detection and analysis mechanisms while agilely deploying new ones. Ruminate does not contribute any new or unique detection mechanisms. It is designed to support existing detection and analysis techniques, such as signature matching, statistical and other forms of anomaly detection, protocol transaction auditing, and dynamic run-time analysis. Ruminate provides the advantage of being able to perform these analysis techniques on payload objects transferred through the network, particularly on objects that can not normally be analyzed with conventional NIDS due to complications such as obfuscation through compression or encoding, high latency analysis, or analysis specific to client application file formats.

In addition to facilitating deep network layer and embedded object analysis in real time, Ruminate provides facilities for extensive transaction auditing through meta-data extraction, normalization, and collection. An important aspect of this meta-data collection is that each service can collect meta-data for the data type that it processes. Therefore, in addition to collecting network layer audit logs, Ruminate collects client application audit logs and can perform expensive operations to collect the required data, such as calculating MD5 hashes of payload objects. Collection of this type of meta-data provides valuable network forensics capabilities, which are vital to incident response efforts. These capabilities also aid in defense efforts against persistent attack groups that are unlikely to be detected by community-wide signatures but historically demonstrate similarities between successive attacks.

# 4 Implementation

Ruminate is implemented as software that runs on commodity hardware and software. Ruminate is open source and will be available for download following publication. Figure 1 shows Ruminate as currently implemented.

The first portion of Ruminate is a component that performs packet capture, stream reassembly, and stream distribution. Packet capture and any other desired packet filtering can be performed at that stage. In addition, Packets are reassembled into streams that are buffered in memory until the stream is closed or terminated for another reason such as a timeout. This component of Ruminate is based leverages heavily the packet processing capabilities of Vortex [5]. Completed network streams are placed into queues by protocol, which can be prioritized if desired. Streams are removed from each queue on a first-in-first-out (FIFO) basis to be processed by the next available application layer analyzer.

Distribution of streams to analyzer nodes is done dynamically using a publish-subscribe model. Network an-

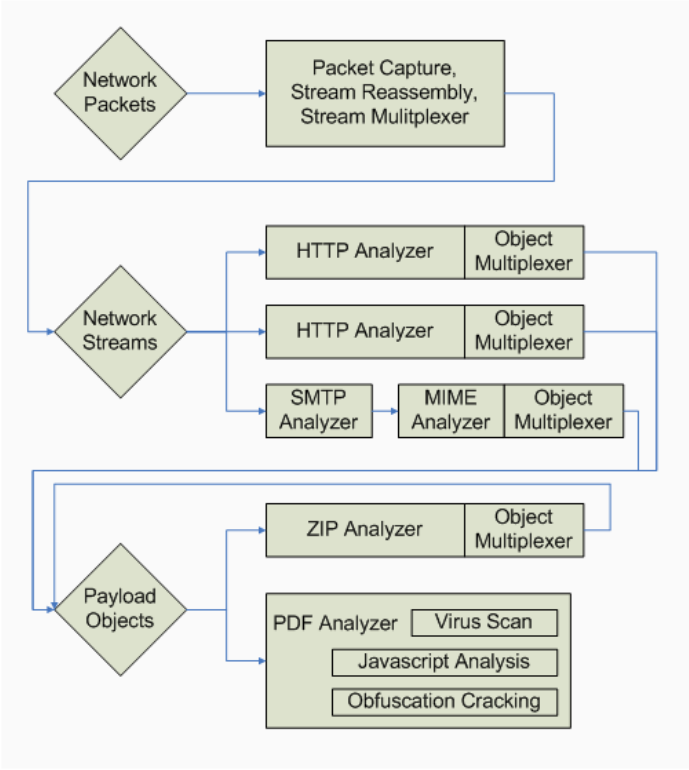


Figure 1: Rumiante Components

alyzers subscribe to feeds of streams using a stream subscription and transfer component. This component subscribes to streams and then transfers assigned streams to shared memory for analysis by the appropriate network protocol analyzer. The stream subscription component operates by connecting to the stream reassembly component and requesting streams of a certain type, ex. HTTP. The stream subscription component indicates to the stream reassembly component when it is ready to receive another stream for analysis. Hence, an analyzer node is not assigned a new stream until it is ready to process it. Network analyzers can arbitrarily connect to and disconnect from the stream reassembly component, making the load distribution truly dynamic. The number of network analyzers can vary over time without affecting the continuous operation of Ruminate.

Under normal operating conditions, a sufficient number of network analyzers are allocated to ensure that streams can always be distributed immediately. In the event that load spikes occur or the number of available network analyzers becomes insufficient, streams are buffered in a FIFO queue. If the size limit for the queue is exceeded, the streams are dropped without being analyzed, similar to packets being dropped by a conventional NIDS. Ruminate does support the pipe-lining of stream subscriptions, whereby a network analyzer can request more than one stream at a time or can request a new stream while simultaneously processing another. In practice, however, this is unnecessary due to the low latency of the high bandwidth links that are likely to be used for transferring bulk network stream data.

Network application layer protocol analyzers process network streams by parsing them to perform actions such as normalization of transactions, generation of transaction events, and extraction of embedded payloads. Audit logs are created for every network application layer event. These transaction events are not audited internally to Ruminate, but the logs are centralized and can be fed directly into log monitoring and correlation engines. Analyzers for HTTP and SMTP/MIME are currently implemented in Ruminate. These protocols were chosen due to their extensive use within most organizations and their preeminence as protocols through which malicious objects are transferred.

The parser for HTTP normalizes and decodes HTTP requests and responses, including all HTTP payload encoding types such as chunked transfer encoding and gzip compression. An event is created for each request/response pair, including the population of data structures that can be used for analysis and event audit. Currently, no standardized mechanisms for defining event auditing criteria exist in Ruminate, but these mechanisms could easily operate on such normalized data structures. The parsing and normalization of network transactions into events that can be audited is similar to the processing required for the detection model of Bro IDS. In Ruminate, an audit log entry is created for each event. This log entry can contain whatever data is desired for auditing. As implemented for demonstration purposes, audit events include the HTTP method, the resource, and various headers (e.g., the Host, Content-Type). Payload objects are decoded from both requests and responses, and they are then submitted to the object multiplexer for further analysis. The event audit logs also contain meta-data pertinent to the payload objects, such as the decoded payload size and the MD5 hash of the payload.

On the other hand, the parser for email is broken into two separate services: SMTP and MIME. These are implemented separately since emails can be transferred through various network protocols. The MIME component can be used in conjunction with other network protocol analyzers. Furthermore, exposing emails as an object to be analyzed allows for the retention of emails, if desired by the organization. As currently implemented, the SMTP analyzer provides minimal functionality. Due to privacy concerns, the SMTP parser does not provide transaction logs containing highly-identifiable personal data, such as email addresses. The MIME parser performs extensive analysis of MIME-encoded emails. Similarly to HTTP, the MIME analyzer provides meta-data (e.g., attachment names, content-types) in audit logs. The MIME analyzer extracts embedded objects in the form of attachments or encoded email data and submits payloads to the object multiplexer for analysis. The MIME transaction event logs include payload-relevant data, such as size and MD5 hashes of embedded payloads.

All embedded payload objects are routed through an

embedded object multiplexer. This component has a common interface used by all analyzers that can extract other embedded objects so that each analyzer uses its own instance of this multiplexer. As such, there is no single bottleneck for embedded objects; the multiplexer receives the objects and meta-data about their origin from each analyzer. It uses libmagic of the file utility [2] to perform inspection of each payload object and determine its type. Based on its determined type, origin, and defined policy, the payload object is then submitted to an appropriate service(s) for analysis. In addition to the disposition of the payload object, the object multiplexer provides audit logs of the result of the payload type determination and other meta-data that can be determined easily (e.g., image dimensions).

To demonstrate the type of embedded object analysis that can be performed, a ZIP archive and a PDF analysis service are implemented in Ruminate. The ZIP archive service was selected because it is one of the most common types of embedded client application objects used to contain other embedded objects. ZIP archives can be used effectively to obfuscate malicious activity in socially-engineered attacks. While the compression and obfuscation of ZIP archives can help attackers evade detection, the container is so popular and widely used that opening a ZIP archive to access malicious content provides minimal obstacle to users who are fooled by social engineering and would have otherwise acted on bare malicious content. The ZIP archive service provides transaction audit data, including the name, size and MD5 hash for every file in the archive. Files extracted from the archive are submitted to appropriate scanning services using the object multiplexer.

The other service currently implemented is a PDF scanning service. PDF documents were selected because of their ubiquitous use across network protocols, as well as the large numbers of existing analysis techniques and recent vulnerabilities. Many trojan PDF documents leverage Javascript to trigger exploits. However, this Javascript is often intentionally obfuscated. The PDF file format allows for various methods to encode or compress document contents, which can serve as an evasion mechanism. These factors, combined with network layer encoding, make detection of trojan PDF documents using traditional NIDS mechanisms very difficult. Thorough analysis of PDFs transferred through the network requires decoding of many layers of encoding, compression, and obfuscation.

The detection mechanisms used in the PDF analysis service include string matching, signature matching, meta-data extraction, Javascript dynamic analysis, and obfuscation-technique cracking. Simple string matching is implemented using the GNU grep utility. More advanced signature matching, some amount of PDF decoding, and some degree of heuristics are implemented using ClamAV [1]. The meta-data contained in PDFs (e.g. document title, author name, PDF creation utility, creation date) are extracted using pdftk [4]. Javascript decoding, dynamic analysis, and other miscellaneous detections are implemented using jsunpack-n [3]. Lastly, XORSearch [7] is used to search for strings obfuscated bu techniques commonly used to evade signature detection (e.g. XOR, ROL).

The services implemented in Ruminate provide valuable analysis capabilities that are not possible within traditional NIDS due to their high latency or high computational expense, as will be shown in Section 5. In addition, the service-oriented architecture of Ruminate allows for a simple integration of existing analysis utilities regardless of attributes of the utility. The programming language in which it was implemented, the native interface, its API or underlying platform do not matter. For example, various components of Ruminate are implemented in the following programming languages: C/C++, Perl, Python, Java, and PHP. Therefore, Ruminate is flexible, allowing the use of any existing implementations of detection mechanisms without adherence to a strict API.

# 5 Experimental Evaluation

To quantify the performance and limitations of our approach, we performed a large set of experiments. First, we compared the dynamic network stream load balancing mechanism implemented in Ruminate with the existing packet load-balancing algorithms used in conventional NIDS. Second, we demonstrated why, in practice, a stream load balance alone cannot perform optimally due to (1) the ubiquitous use of network layer encoding schemes and (2) the fact that variance in the computational complexity that stream load balance entails can create very disparate traffic load on the analysis hosts. This stems from the fact that we are interested in performing embedded object analysis that goes beyond simple pattern matching. Third, we measured both the performance overhead and detection capability from the application of Ruminate on the traffic collected real-time from a live network feed.

The data we used for the empirical evaluations of Ruminate was collected from the Internet router of a university campus with a student body of 30,000. More specifically, the empirical evaluations comparing connection load balancing techniques used a packet trace that spanned a 24-hour period from midnight to midnight, local time, and consisted of 973,718,083 packets totaling 737 GB. This packet trace was replayed for each permutation of the stream load balancing parameters. Section 5.2 refers to a collection of all PDF documents that were transferred through the network during a time period of about three days; this corpus comprises 44,921 PDF documents that total 24 GB of data. All evaluations that demonstrate detection capabilities, the type of objects transferred through the network, and performance

Table 1: Specification of Servers Used in Evaluation

| Server Role | Total | CPU Type | CPU Speed | Cores | RAM |
|---|---|---|---|---|---|
| Front End | 1 | intel X5550 | 2.67GHz | 16 | 72 GB |
| Analysis | 4 | intel X3363 | 2.83GHz | 4 | 8 GB |

of the system were taken from a two-day observation of Ruminate operating on live traffic. The approximate volume of this two-day live run was 1.6 billion packets and 1.4 TB of data. Since Ruminate currently implements HTTP and SMTP protocol parsing, only TCP ports 80 and 25 were analyzed. (Counts above only include this filtered data). The potential limitations of the data used are addressed in Section 6.

We deployed our system prototype on a cluster of five servers, the specifications of which are outlined in Table 1. The front end node was used for packet capture or for replay and stream reassembly. Protocol analysis was performed on the analysis nodes. Due to an abundance of excess capacity, embedded-object analysis and log centralization were performed on the front end node. The servers were connected with Gigabit Ethernet. In all servers, we used Linux for the operating system.

## 5.1 Connection Load Distribution

We conducted experiments to measure the load balancing behavior of the dynamic stream-based load balancing over static hashing of packet header data. In the latter case, a static hash function applied to packet header data is deterministic. Simulating the load distribution based on static packet header hashing was performed by operating on network flow records. Dynamic load balancing distributes load based on the availability of analyzer nodes. which varies based on the load experienced by each analysis node and is not deterministic. Therefore, the empirical data for dynamic load distribution was obtained through actually distributing load to analysis nodes and varying the parameters, such as the number of nodes between replays of the same traffic.

To demonstrate the characteristics of packet level load balancing, network flow data was extracted from the packet capture trace using Argus. Load balancing was simulated by distributing load (i.e. the amount of network data in flow record) across analysis nodes using a hash function on network and transport layer parameters. A 5-tuple hash was used which included the source IP, source port, destination IP, destination port, and transport layer protocol (e.g., TCP) of the packets. MD5 was used as the hash function. Load statistics were recorded using measurement windows of 1s, 3s, and 5s. The byte counts from the network flow data were scaled down by a constant factor of 0.41 to allow the statistics from these calculations to be directly comparable to those gathered during the trials using dynamic stream load balancing. Reasons for this data reduction include differences in data sizes for packets and reassembled payload data (due to a lack of packet headers, retransmissions only being counted once, etc.) and to compensate for the inability to fully reassemble some streams (due to lost packets, imperfections in TCP reassembly routines, etc.). In all cases, these parameters were chosen to represent the best possible scenario for packet header hashing. For example, 5-tuple hash was used instead of 2-tuple hash, and MD5 was used as the hash function instead of a weaker hash function.

While packet header hashing is static, dynamic stream balancing results in streams being multiplexed to analysis nodes based on availability. Full-length simulations were run to accurately simulate dynamic stream-based load balancing. The packet capture, stream reassembly, and stream distribution functionality of Ruminate were used in these evaluations. The packet captures are replayed in real time on the front end node, which reassembles and buffers streams then passes finished streams to one of the analysis nodes based on their availability. The analysis nodes simulate processing of the streams by downloading the stream content and then processing the stream data. In lieu of performing any actual processing on the stream data, analysis was simulated by sleeping a length of time commensurate with the size of the stream, such that analysis was upper-bounded by an arbitrary speed specified for the trial. Each analysis node had the same maximum processing speed of approximately 100 Mbps per total number of analysis nodes in the trial. The number of analysis nodes is held constant throughout a given trial. In trials involving more than four analysis nodes, multiple independent instances of the analysis node software are run on a single physical analysis server.

Figures 2 and 3 portray the results of simulations using eight analysis nodes. These graphs depict the maximum and mean load (measured in Mpbs) over the time of the trial. Figure 2 shows statistics from 5-tuple static packet load balancing. Figure 3 shows statistics from stream based dynamic load balancing. The lines represent the mean bandwidth each node is assigned at a resolution of one hour. The points represent the maximum load assigned to a node for a single measurement window of five seconds during a given hour. Each node is represented by a point of unique shape. Figures 4 and 5 illustrate the same results when 64 analysis nodes were used instead of 8.

Figure 6 shows the overall traffic for this packet trace. The average total payload bandwidth for this packet trace is 30 Mbps, but this number increases to about 65Mbps during the busiest portion of the day. It peaks at 180 Mbps for a single 5s interval, but there are very few 5s intervals when payload data over 120 Mbps is observed. Note that despite static load distribution be-

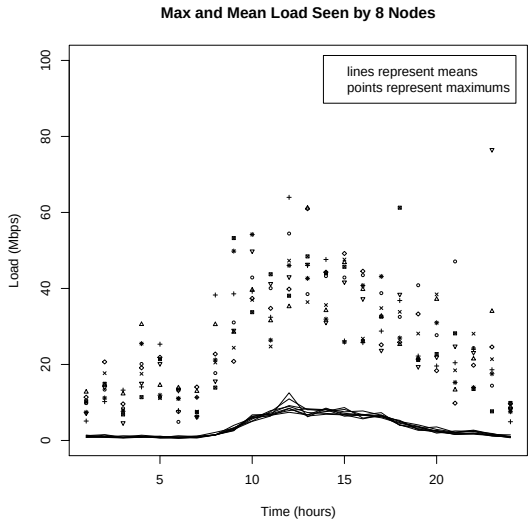


Figure 2: Static Load Balancing (5 second window)

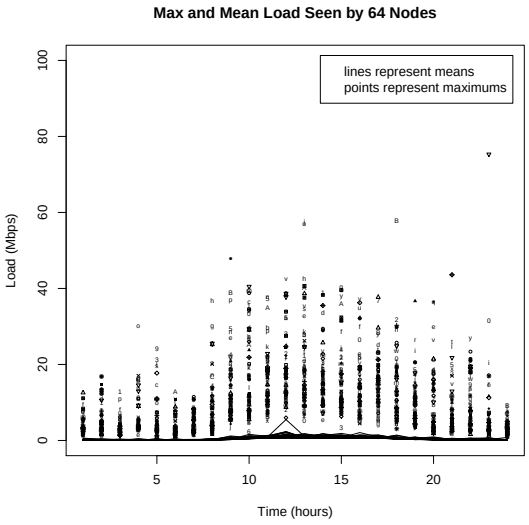


Figure 4: Static Load Balancing

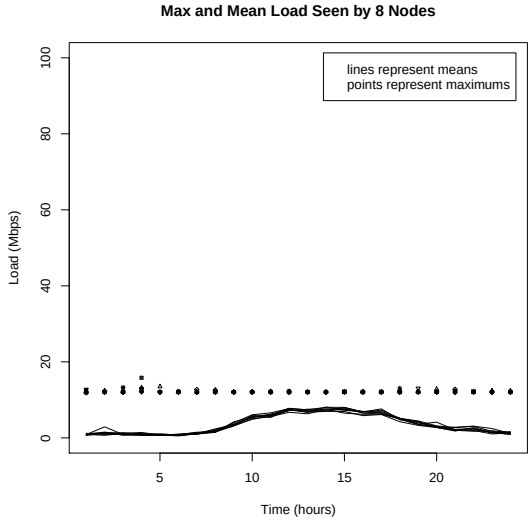


Figure 3: Dynamic Load Balancing

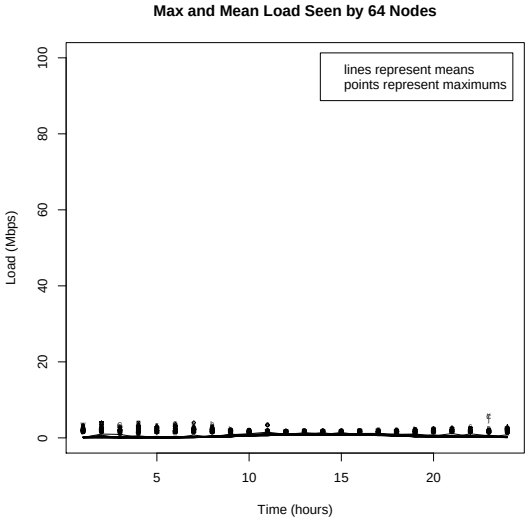


Figure 5: Dynamic Load Balancing

tween eight analysis nodes, it is common for a single node to be assigned load up to and exceeding 50 Mbps. Furthermore, when the number of analysis nodes is increased to 64, there are still single analysis nodes assigned loads up to and exceeding 50 Mbps. Conversely, in the dynamic load balancing trials, the load is distributed close to the desired rate of 100 Mbps divided by the number of nodes. For example, with eight analysis nodes, the upper bound of the load assigned to any node is close to their individual capacity of approximately 12.5 Mbps. Similarly, with 64 analysis nodes, the maximum load assigned to a given node remains under approximately 2 Mbps as desired.

Figures 7 and 8 demonstrate the maximum load seen by any one node during a time window of the specified length. The super-imposed line reflects the curvature of an ideal distribution (max bandwidth/number of nodes). Note that in both cases of packet header load balancing, the maximum load drops off dramatically at first, closely following the ideal line. In the case of static load balancing, the maximum load line is basically horizontal after eight nodes. On the other hand, in the case of dynamic stream load balancing, the maximum load scales with the number of nodes.

The results show that packet header load balancing, while effective for a small number of nodes, can not scale to very large numbers due to the variance in load assigned to a single node. However, dynamic stream load balancing provides very efficient scaling of analysis to a large number of nodes in a NIDS cluster.

In practical terms, the use of static hashing means adding more nodes to a NIDS cluster provides dimin-

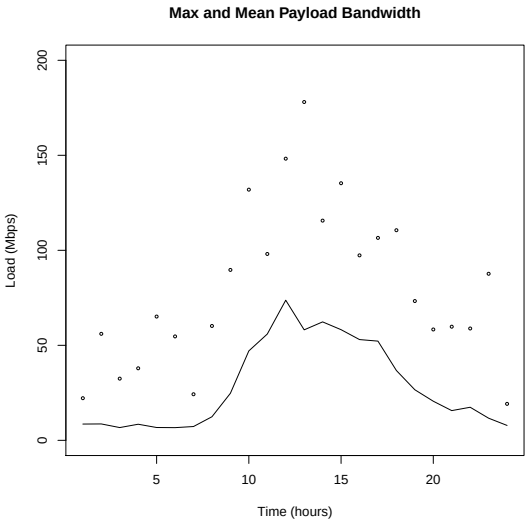


Figure 6: Overall Average Bandwidth

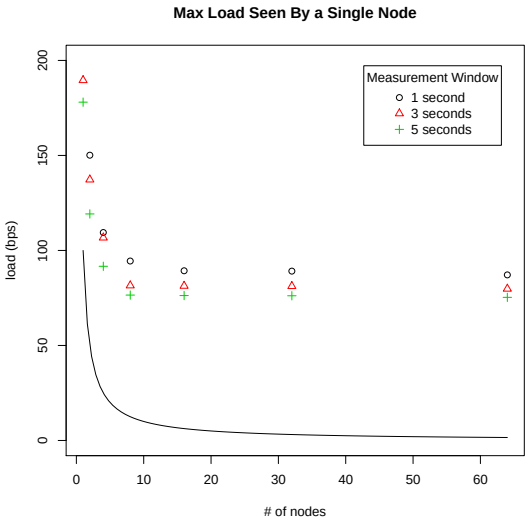


Figure 7: Maximum Load–Static Distribution

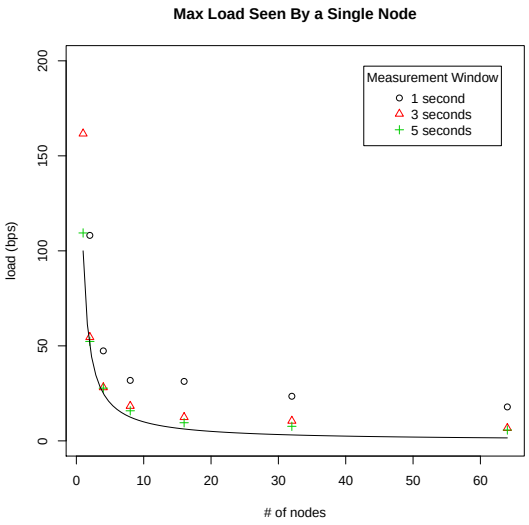


Figure 8: Maximum Load–Dynamic Distribution

ished returns. A single node still has to be able to process the maximum bandwidth that could be assigned to it, or large amounts of packet loss must be accepted, even if, on average, a given node processes much less. Static header hashing can not scale to highly-parallel clusters to perform computationally-expensive analysis or leverage many processors with low individual capacity. However, the dynamic stream based load balancing of Ruminate provides a high level of scalability.

## 5.2 Variance in Analysis Techniques

Distributing load in a NIDS is further complicated when the complexity or latency of the analysis performed by the NIDS varies greatly. Conventional NIDS perform analysis that has relatively low variance (e.g., signature matching). Ruminate is designed to efficiently support analysis across a wide range of latencies and computational complexities.

Many conventional NIDS are limited in their ability to effectively process embedded objects due to network protocol encoding such as HTTP chunked encoding, HTTP compression, or MIME encoding such as base64. Decoding these network protocol encoding introduces significant intra-packet processing or state tracking, computational complexity, and latency. However, these encoding mechanisms represent a very large portion of the embedded objects transferred through the network. Figures 9 and 10 demonstrate the observed network layer encoding used for both HTTP and MIME during the two day live run. These figures show the decoded volume of payload objects transferred by encoding type. The volume presented in these graphs is a reflection of the size of decoded payload objects. Therefore, transactions that do not contain an embedded payload, such as an HTTP request/response that results in a 404 error, are considered to have an embedded object volume of 0. Note that for HTTP, encoded payloads make up a significant portion of the embedded object volume and that for MIME, encoding methods such as quoted-printable and base64 are used to transport the the vast majority of the payload objects by volume. All of these encoding methods can and do inhibit the detection capabilities of traditional NIDS.

The network protocol decoding and the analysis in the PDF scanning service demonstrate the amount of variance that Ruminate is designed to support. Table 2 shows the time required and resulting analysis bandwidth for various decoding/decompression mechanisms and analysis methods performed on a corpus of PDFs collected from the campus network. For each analysis method, the corpus was analyzed serially. The time to analyze the corpus was recorded and the aver-

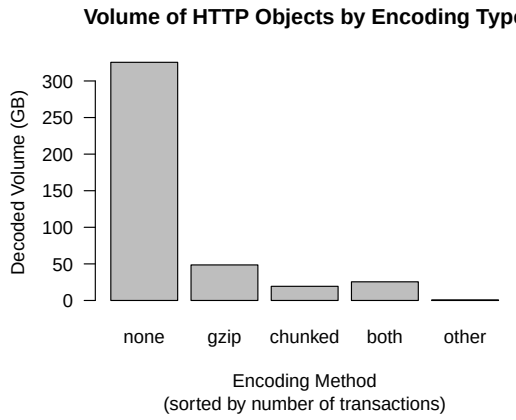


Figure 9: HTTP Encoding Observed

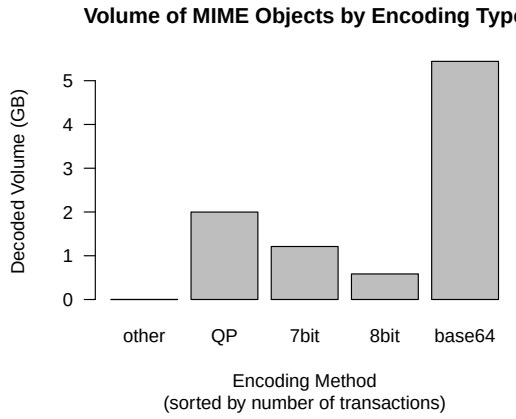


Figure 10: MIME Encoding Observed

age bandwidth of the technique was calculated. Tests were rerun at least three times, ensuring consistency in measurements to at least the two significant digits presented. While detailed performance data is not presented here, it is noted that all analysis techniques were CPU-bound except for meta-data extraction, which was IO bound. In the case of the decoding mechanisms, namely gzip decompression and base64 decoding, the bandwidth is calculated from the decoded volume. The standard Unix utilities gzip and base64 were used for these measurements. Note that in the case of base64 decoding, the documents were encoded with base64, but newline characters were inserted every 60 characters to match the encoding found in MIME. Javascript analysis was only performed on PDFs that contained markers for Javascript streams as determined by a simple string match. There were only 746 PDFs, totaling 350 MB, on which Javascript analysis was performed. Therefore, an average bandwidth of 3.3 Mbps was calculated for the whole corpus, but since Javascript analysis was only performed on a subset, the average bandwidth of the mechanism is actually 0.048 Mbps on the subset of PDFs with markers for Javascript.

Table 2: Variance in Embedded Object Analysis

| Analysis/Decoding Method | Time (min) | Average Bandwidth (Mbps) |
|---|---|---|
| String Match | 0.23 | 14000 |
| gzip Decompression | 3.0 | 1100 |
| base64 Decoding | 6.2 | 510 |
| Virus Scan | 19 | 170 |
| Meta-data Extraction | 67 | 49 |
| Javascript Analysis | 970 | 3.3 |
| Obfuscation Cracking | 2100 | 1.5 |

We have shown that a large portion of embedded-object volume involves some sort of network protocol encoding on a real network. We have also demonstrated the speed of some analysis techniques that could be performed on a given embedded object type using data from a real network. There was a large variance in the average bandwidth at which each of these techniques could be performed. The variance in speed was four orders of magnitude. Furthermore, some analysis technique operate at speeds far below the bandwidths at which detection mechanisms must operate in traditional NIDS. These techniques, however, represent the type of detections that Ruminate is designed to support.

## 5.3 Analysis of Detection Capabilities

The primary contribution of Ruminate is to provide a framework that can efficiently utilize commodity hardware and software to perform analysis of network protocols and embedded objects including recursive analysis not possible in traditional NIDS architectures. While no original detection mechanisms are presented, Ruminate is leveraged to apply network analysis and payload analysis techniques that can not be preformed efficiently in traditional NIDS. The analysis techniques demonstrated include valuable detections on real network traffic.

An important capability provided by Ruminate is the parsing of network protocol transactions and embedded objects to generate transaction events which can be audited. Transaction events are generated by each analysis service. The HTTP, MIME, Object Multiplexer, ZIP, and PDF analysis services all generate audit events for the protocol/file format they analyze. These audit events include data that is relatively computationally cheap to generate, such as the URL of the HTTP events, and data that is expensive such as MD5 hashes of decoded payload content. Ruminate demonstrates this capability through generation of transaction event logs which are centralized for potential auditing and correlation.

Table 3 shows the number of audit events generated

Table 3: Audit Event Counts by Analysis Service

| Analysis Service | HTTP | Object Multi-plex | MIME | ZIP | PDF |
|---|---|---|---|---|---|
| Audit Event Count | 22,300K | 19,000K | 900K | 394K | 30K |

by Ruminate during the two-day live traffic processing. These audit logs are important because they represent a large amount of computationally expensive analysis that must occur to parse and normalize the network transactions and payload objects. The traffic patterns on this day were very similar to those of one day packet trace as shown in Figure 6. The overall packet capture bandwidth, including packet headers, during that time period was 60 Mbps. Nearly 500 GB of embedded payload objects were extracted and analyzed during that time period. No packets were dropped by the Ruminate system.

With Ruminate, it is possible to collect extensive transaction audit data, including network protocol and embedded object meta-data. Figures 11 and 12 show the content type of embedded objects as indicated in the "Content-Type" headers for the specified protocols. Note that any arbitrary protocol or embedded object file format meta-data could be exposed in event audit logs.

In addition to collecting extensive audit logs, Ruminate can also perform extensive analysis that results in detections not possible with conventional NIDS. The PDF scanning service provides detections of malicious PDF documents using numerous methods.

During the time in which these detections occurred, packet capture and stream reassembly were preformed on the front end server as defined in Table 1. All network protocol analyzers were run on the four analysis nodes. Four instances of the HTTP analyzer and one instance of the SMTP/MIME analyzers where run on each analysis node. The PDF and ZIP analyzers were run on the front end server due to an abundance of available resources, but they could be run on any set of servers. Event audit logs were centralized to the the front end server also.

Performance statistics were kept during this period also. The statistics gathered were CPU utilization and memory consumption. In most cases the performance statistics represent the combination of multiple threads/processes performing the same functionality. The performance was monitored at a resolution of 1 second, but has been averaged over 1 hour intervals for presentation. Figure 13 shows CPU resources used by the front end to perform packet capture, stream reassembly, and stream distribution (Stream Reassembly.) and the CPU time used to collect and store the event audit logs generated by Ruminate (Log Centr.). Note that that Ruminate currently uses a maximum of about 25% of one processor

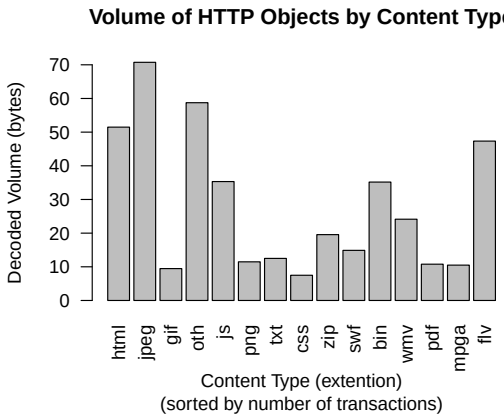


Figure 11: HTTP Content Type Observed

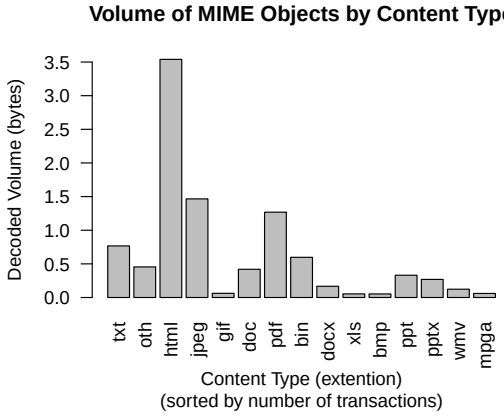


Figure 12: MIME Content Type Observed

core. This load is an aggregate of two processes for the stream reassembly component (one for HTTP streams, and one for SMTP streams) which both use utilize multiple threads, as well as other processes on the system, namely audit log collection.

Figure 14 shows the memory usage of the packet capture and stream reassembly. During operation, the stream reassembly and distribution component of Ruminate consumes between 6% and 7% of the front end node's memory, or about 5 GB. The majority of this memory is consumed in large data structures for tracking connections which is why the memory consumption does not dip when the load does during the night of the first day and morning of the second day and why the memory consumption does not rise as much as load during high traffic times. A portion of this memory is also used for collecting network streams and buffering them before distribution to the analysis nodes. This is the only component of Ruminate using a pertinent amount of memory, so no other memory consumption statics will be presented.

Streams are transferred from the stream reassembly component to an available network protocol analyzer.

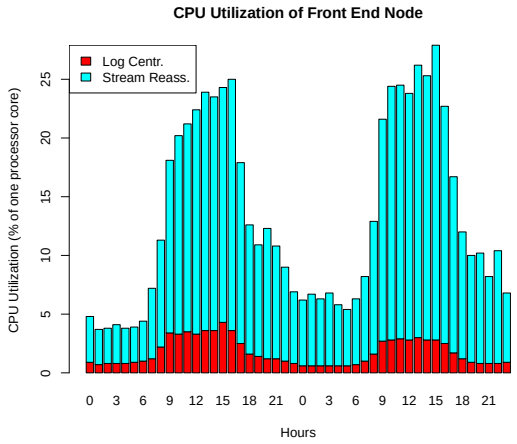


Figure 13: CPU Utilization on Front End Server

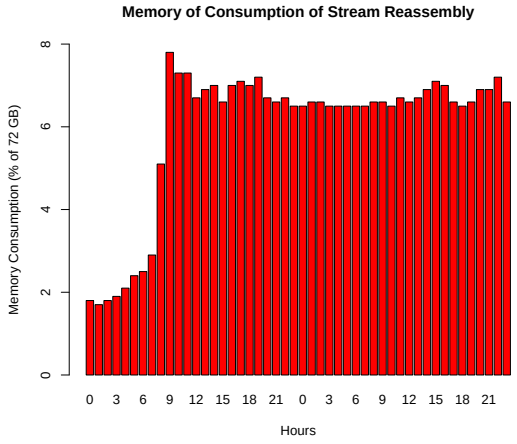


Figure 14: Memory Consumed in Stream Component

The protocol analyzers were run on four similar analysis nodes. The load characteristics on all four nodes were very similar. Figure 15 shows the components on the analysis nodes which used a non-negligible amount of CPU time. Note that the SMTP analyzer is intentionally excluded because it used consistently under 0.1% CPU time. As currently implemented, the SMTP analyzer does a very limited amount of processing. The CPU consumption on the analyzer node is dominated by the HTTP analyzer (HTTP Anal.). The MIME analyzer (MIME Anal.) uses such a small amount of processing power because the amount of SMTP traffic is considerably smaller than HTTP traffic. The measurements for the stream subscription and transfer (Stream XFER) component and the object multiplexer (Object MUX) components include the load contributed for both the HTTP and SMTP traffic. While minimal, CPU utilization of the stream subscription and transfer component provides a good measure of the amount of overhead incurred by distributing load across the network as this component merely subscribes to the stream distribution component and copies assigned streams into shared memory for analysis by the appropriate protocol analyzer. The memory consumed by protocol analysis was minuscule, never topping 100 MB. Note that all of the aggregate load on the analyzer nodes utilized at most 1/3 of one of the four CPUs in these systems.

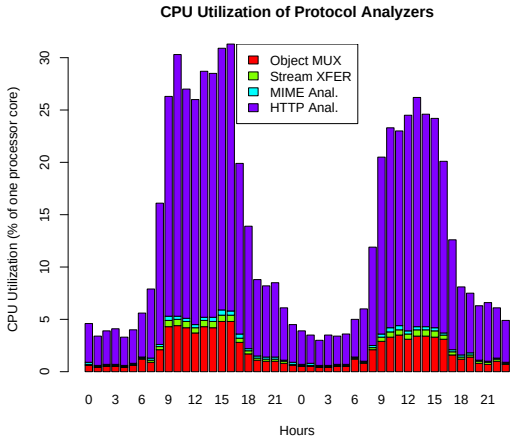


Figure 15: CPU Use for Analyzer Nodes

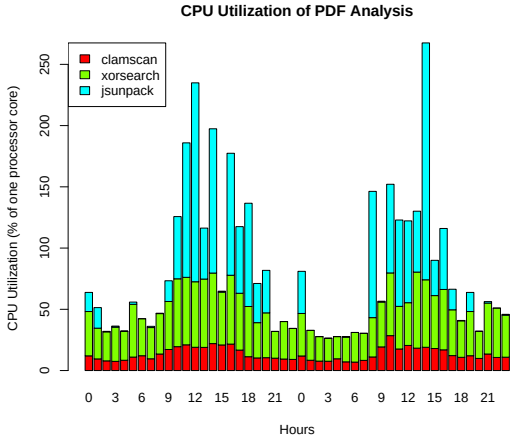


Figure 16: CPU Use for PDF Service

While run on the front end node as currently implemented, the performance statistics for the PDF analysis service will be presented separately. Figure 16 shows the CPU usage of the PDF service. Only the three computationally expensive sub-components of the PDF analysis service are shown as all other portions of the PDF service consume orders of magnitude less CPU time. The virus scanning (clamscan) and obfuscation cracking (xorsearch) provide a relatively constant load. Since the Javascript analysis (jsunpack) is only performed on PDFs with Javascript and because the analysis performed on individual PDFs is highly dependent on the specific data transferred through the network, there is great variance in the load of the that sub-component. Note that while PDFs comprise less than 3% of the objects extracted from

the network for analysis, the analysis performed on them requires greater CPU resources than all other analysis performed in Ruminate combined. This is precisely the type of expensive but valuable detection mechanism Ruminate is designed to make efficient and scalable inside of NIDS. No data is presented for the ZIP analysis service because it used an extremely low amount CPU time.

# 6 Discussion

## 6.1 Latency

The majority of NIDS are designed to be able to analyze packets and detect real-time attacks in order to support prevention through actions on the network traffic (e.g., dropping packets that contain attacks). The drivers for the capabilities unique to Ruminate differ significantly with these real time constraints. Ruminate can perform many types of analyses with higher efficiency since it places higher priority on auditing and detection capabilities than on prevention capabilities. For example, performing dynamic analysis or other high-latency analyses may not be compatible with prevention within some protocols (e.g., HTTP) due to untenably high latency. Furthermore, the focus of forensics or historical detection capabilities (e.g., collecting network transaction audit logs that include embedded object meta-data) does not benefit by the real-time constraints imposed by prevention.

Notwithstanding the low priority placed on prevention in its design and testing, Ruminate can be utilized to provide prevention capabilities, especially for protocols that allow for significant delay (e.g., SMTP). This is especially true of the portions of Ruminate that do not perform high-latency analysis. For example, Ruminates stream reassembly and buffering mechanism, which incrementally collects but does not provide the stream data to external analyzers until the stream is finished, introduces from 1 to 2 ms of delay from the time the stream is closed until the data is provided to an external analyzer.

## 6.2 Memory Usage

Ruminate makes extensive use of memory for buffering. For example, incomplete streams are buffered in memory. This buffering facilitates simplicity in detections on embedded objects and efficient utilization of processors. Memory usage is not deterministic. It is based on traffic and payload characteristics. Memory use increases with the size of network streams and embedded objects and the length of time these objects are collected or processed.

Table 4 shows the distribution of network connections and volume by connection length for the same data set used throughout this paper.

Table 4: Distribution of Connections by Length

| Length (s) | [0,10) | [10,100) | [100, 1000) | [1000,*] |
|---|---|---|---|---|
| Count | 10594342 | 921972 | 309567 | 23607 |
| Volume | 186GB | 113GB | 111GB | 58GB |

Given typical network traffic, this use of memory is not a serious limitation. While memory use depends on many factors, it is not uncommon for Vortex IDS, on which the stream reassembly component of Ruminate is based, to require between 1 GB and 10 GB of RAM monitoring links up to 1 Gbps. Due to the practicality and economy of equipping commodity servers with large quantities of RAM, decreasing the amount of RAM consumed has not been addressed. It is recognized that it could be possible to create a resource exhaustion attack against Ruminate by causing large network streams to occur which stay open for long periods of time. Ruminate does provide mechanisms for limiting usage of RAM through limits on the amount of RAM to be used by individual streams, timeouts on streams that are not closed but which don't transfer data, etc. Attacks that attempt to overwhelm Ruminate through excessive RAM utilization would be noisy due to anomalous network parameters and expensive for attackers to perpetuate. Further possible improvements are detailed in Section 7.

## 6.3 Evasion Techniques

A major motivation for Ruminate is to be able to perform analysis on client application objects transferred through the network, including recursive analysis of embedded objects. For example, a malicious payload can often evade detection by NIDS by merely being wrapped in a container such as a zip archive. It is conceded that Ruminate's effort to recursively decode objects, or even the highly computationally expensive analysis that Ruminate can do but which can not be predicted in advance, make it vulnerable to resource exhaustion, depending on conditions, some of which are in control of the attacker. Attempts at resource exhaustion, such as transferring an archive that contains very many layers of nested archives or transferring documents with computationally expensive but benign scripting could be made, and some may be practical. It should be noted however, that this can be mitigated through proper limits on recursion and individual analysis as well as detection of apparent resource exhaustion attacks.

## 6.4 Data Limitations

One limitation to empirically evaluating Ruminate is the amount and quality of data used. The amount of data traffic on the university network is not large enough to stress the capabilities of the hardware on which Ruminate is currently deployed. The volume of traffic

could be scaled up nearly an order of magnitude without any changes to the implementation evaluation. While greater overall volume would be useful in proving Ruminates scalability, the current volume is large enough for demonstration purposes. A serious limitation of the data used in these evaluations is that many packets are dropped in the tapping infrastructure and link that transfers packets from the main Internet router to the lab in which Ruminate is deployed. These packet drops that occur externally to the Ruminate system can be inferred by observing holes in packet streams. Nearly 1% of packet loss was inferred to have occurred externally to Ruminate in both the day-long packet trace and the two-day live trial. The effects of this packet loss were magnified by Ruminates inability to reconstruct full streams if holes existed in the streams. While Ruminate would have been able to reconstruct more complete stream and embedded object data, the presence of this packet loss did not significantly affect the validity of the empirical evaluations based on the streams that could be reconstructed. It is possible that other limitations of Ruminate, such as its inability to correctly emulate all TCP/IP stacks, also contributed to the low volume of reassembled stream data. However, these limitations did not affect the validity of the data that was correctly reassembled.

# 7 Future Work

Many facets of Ruminate could potentially be researched further, and numerous implementation weaknesses could be improved. For example, the stream re-assembly component based on Vortex has some limitations in its ability to correctly model all TCP implementations, as well as in its tolerance for packet loss. Ruminate currently provides a flexible platform on which many potential detection mechanisms could be implemented. Integrating additional detection mechanisms (e.g., methods found in conventional IDS, mechanisms not possible in current NIDS) would make Ruminate more valuable.

Another area of improvement for Ruminate is that it currently only processes network transaction events after the transport layer stream is finished. While this facilitates efficient dynamic load balancing, the latency of analysis could be improved if portions of streams representing individual network application layer transactions were distributed to an available analyzer node before the stream terminates. In addition, Ruminate provides effective dynamic load balancing, including the ability to dynamically add additional resources in the form of more analyzer nodes. However, we do not provide a comprehensive study of multiple optimization objectives for the dynamic allocation of resources beyond mere provisioning. Such study could improve the overall system efficiency and is a natural topic for future work.

# 8 Conclusions

In this paper, we present Ruminate, an object-centric network payload analysis architecture. To that end, Ruminate aggregates and re-orders traffic packets into streams. Then it employs a modular consumer-producer mechanism to outsource the computationally expensive tasks to other machines. These machines run protocol parsers, object decapsulation, and object analysis engines to ferret-out attacks that are concealed deeply in network payload objects.

Furthermore, Ruminate allows the traffic to be prioritized and treated preferentially based on the reassembled content rather than mere packet header and protocol information. Our experiments, using real-world traffic from a deployment on a site with 30,000 users involving 973,718,083 packets and totaling 737 Gb, showed that we can dynamically process and load-balance the analysis of multiply encoded and embedded desktop application objects, such as Adobe PDF documents or ZIP archives, scalably and without packet loss.

# References


[1] Clam AntiVirus. http://www.clamav.net/.

[2] Fine free file command. http://www.darwinsys.com/file/.

[3] jsunpack-n. https://code.google.com/p/jsunpack-n/.

[4] pdftk. http://www.accesspdf.com/pdftk/.

[5] Vortex. http://sourceforge.net/projects/vortex-ids/.

[6] VRT razorback. http://labs.snort.org/razorback/.

[7] XORSearch. http://blog.didierstevens.com/programs/xorsearc

[8] D. Antoniades, M. Polychronakis, S. Antonatos, E. Markatos, S. Ubik, and A. Øslebø. Appmon: An application for accurate per application network traffic characterization. In *In IST Broadband Europe 2006 Conference*. Citeseer, 2006.

[9] N. Borisov, D. Brumley, H. Wang, J. Dunagan, P. Joshi, C. Guo, and I. Nanjing. A generic application-level protocol analyzer and its language. In *14h Symposium on Network and Distributed System Security (NDSS)*. Citeseer, 2007.

[10] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):5–16, 2007.

[11] N. W. Daniel Geer. Advanced persistent threat, 2010. `http://www.networkworld.com/news/tech/2010/041210-tech-update.html`.

[12] L. Deri and F. Fusco. Exploiting commodity multicore systems for network traffic analysis. http://luca.ntop.org/MulticorePacketCapture.pdf, Aug. 2009.

[13] R. Dhamankar, M. Dausin, M. Eisenbarth, and J. King. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/, Sept. 2009.

[14] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX Security Symposium*, 2006.

[15] R. Duncan and P. Jungck. packetC language for high performance packet processing. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 450–457, 2009.

[16] R. Gopalakrishna, E. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Security and Privacy, 2005 IEEE Symposium on*, pages 18–31, 2005.

[17] F. Gu, Z. Zhang, and L. Wang. A Trace-Driven simulation study of load balancing on distributed NIDS. In *Networking, Architecture, and Storage, 2008. NAS '08. International Conference on*, pages 159–160, 2008.

[18] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference*, pages 1–12. ACM, 2008.

[19] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350. ACM, 2006.

[20] W. Lian, F. Monrose, and J. McHugh. Traffic classification using visual motifs: an empirical evaluation. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, pages 70–78, New York, NY, USA, 2010. ACM.

[21] P. Lin, Y. Lin, T. Lee, and Y. Lai. Using string matching for deep packet inspection. *Computer*, 41(4):23–28, 2008.

[22] Mcafee Labs. McAfee, Inc. 2010 Q3 Threat Report Reveals Average Daily Malware Growth at an All Time High. http://www.mcafee.com/Q3_Threat_Report, Nov. 2010.

[23] T. Nguyen and G. Armitage. A survey of techniques for Internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2009.

[24] M. Norton, D. Roelker, and D. R. S. Inc. Snort 2.0: High performance multi-rule inspection engine.

[25] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300, Rio de Janeiro, Brazil, 2006. ACM.

[26] V. Paxson. Bro: a system for detecting network intruders in real-time. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.

[27] V. Paxson, R. Sommer, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *Sarnoff Symposium, 2007 IEEE*, pages 1–7, 2007.

[28] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In C. Krügel, R. Lippmann, and A. Clark, editors, *RAID*, volume 4637 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2007.

[29] Z. Ryan Naraine. Adobe warns of flash, pdf zero-day attacks, 2010. `http://www.zdnet.com/blog/security/adobe-warns-of-flash-pdf-zero-day-attacks/6606`.

[30] R. Smith, C. Estan, and S. Jha. XFA: faster signature matching with extended automata. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 187–201, 2008.

[31] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *ACM SIGCOMM Computer Communication Review*, 38(4):207–218, 2008.

[32] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, New York, NY, USA, 2003. ACM.

[33] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. *Security and Privacy, IEEE Symposium on*, 0:305–316, 2010.

[34] L. Tan and T. Sherwood. Architectures for bit-split string scanning in intrusion detection. *Micro, IEEE*, 26(1):110–117, 2006.

[35] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection*, 2007.

[36] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[37] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*. Citeseer, 2008.

[38] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, San Jose, California, USA, 2006. ACM.