A Framework for Collection and Management of Intrusion Detection Data Sets

Benjamin D. Uphoff
Los Alamos National Laboratory
bduphoff@lanl.gov

Paul J. Criscuolo
Los Alamos National Laboratory
pcrscl@lanl.gov

Abstract

Two areas in intrusion detection research receive little attention: data collection and data management. Gigabit Ethernet is becoming widely deployed, with ten gigabit Ethernet not far behind. Many current solutions strain under such bandwidth rates, resulting in data loss. This is unacceptable for accurate, reliable intrusion detection systems. Data management solutions vary greatly from product to product. Typically, older data is periodically migrated to some archived format. Once archived, the data set cannot be easily queried or analyzed without being imported back into the original tool. This makes forensics and trend analysis extremely difficult.

This paper addresses data collection and management for intrusion detection by providing a framework designed to accommodate high-volume, heterogeneous data sets. This framework solves many of the problems of conventional approaches to intrusion detection. Distributed computing is leveraged to assure scalability. Data can be captured, queried and analyzed in real-time; data set sizes are limited only by available storage. Benchmarks of the initial prototype are also provided.

1 Introduction

Two areas in intrusion detection research receive little attention: data collection and data management. These areas have become increasingly important with the prevalence of Gigabit Ethernet. Ten gigabit Ethernet is also growing in popularity, resulting in even larger data sets to collect and manage. Current solutions deployed in large networks strain under such bandwidth rates, resulting in data loss and potentially missed intrusion events. This is unacceptable for accurate, reliable intrusion detection systems (IDSs).

In terms of data management, solutions vary greatly from product to product. Often, older data is archived to ensure adequate performance and minimize storage overhead. Once archived, it becomes difficult to query and analyze the data, making forensics and trend analysis extremely difficult.

This paper addresses data collection and management for intrusion detection by providing a framework designed to accommodate high-volume, heterogeneous data sets. This framework solves many of the problems of conventional approaches to intrusion detection and network traffic data collection. Distributed computing is leveraged to assure scalability. Data can be captured, queried and analyzed in real-time; data set sizes are limited only by available storage.

The remainder of the paper is as follows: Section 2 establishes the motivation behind the research and summarizes the contributions. Section 3 discusses related work in the area. Next, section 4 describes the intrusion detection framework and the various modules therein. Section 5 gives a high-level description of the implementation being developed at Los Alamos National Laboratory. Section 6 provides some preliminary benchmarking results. Section 7 discusses future work while section 8 concludes the paper.

2 Motivation and contributions

The motivation for this work was provided by the network security demands at Los Alamos National Laboratory (LANL). At LANL, a network intrusion detection system (NIDS) is used to monitor the laboratory's high throughput network. Data collected by the NIDS is similar to technologies such as Cisco's NetFlow and Riverstone's LFAP, reducing hundreds of gigabytes of packet data into a more manageable size. Like these other products, the NIDS collects data on source and destination IP addresses, source and destination ports, byte totals, timestamps and various supporting flags. On a given day at LANL, the NIDS can capture as many as 30 million records for a total of over three gigabytes of data.

The NIDS data set is used to perform long-term analysis on network activity along with network forensics. To achieve this, the data is often analyzed with custom Perl scripts and more rudimentary tools like grep. When working with large subsets of the data, this process can be very inefficient. To avoid this problem, analysis typically begins with a "data slicing" operation. These operations are typically of the form *get all record for IP address a.b.c.d for May* or *get all port xyz traffic for the last six months*. After performing the initial operation, the resulting subset is analyzed to answer more complicated questions about the data set. With a reduced data set, custom Perl scripts can be very effective analysis tools.

Given the nature of the data analysis being performed, it became clear that the data-slicing problem was the most pressing issue to be resolved. A flexible, extensible data management solution was needed. The first and most obvious solution was to place data in a relational database. Clearly, the data-slicing problem would be solved by this approach. However, there were some fundamental drawbacks to this solution. One gigabyte of network traffic data does not translate into a one-gigabyte relational database. In most cases, raw data formats will result in a smaller footprint when compared to a relational database. Section 6.2 examines this issue in detail. Although the cost of storage continues to drop, one of the main design goals was to avoid keeping redundant copies of data. Using a relational database would result in keeping two copies of the data: one in relational tables and another in raw format. This was not acceptable due to the economic reality of maintaining multi-terabyte data sets.

In the end, it was decided that the raw data files would be indexed on several key fields commonly used in data slicing operations. The index structure chosen was a b+ tree with fully filled pages. This data structure provides fast look-ups and low overhead. Each page in the tree if fully filled, because the index is bulk loaded and no inserts will ever be made, as the data is static. The end result is a small, fast mechanism for information retrieval.

One last motivator for this work was the desire to have instantaneous access to records gathered by the NIDS. In the past, this data would be made available in six-hour segments, which would sometimes have a lag in processing of twelve hours. Thus, it was impossible to analyze network activity in real-time using this data set. To solve this problem, records can be streamed to the system as they are recorded by the NIDS. The incoming data items are stored and indexed using Berkeley DB. These dynamic indices are periodically converted into static b+ trees for permanent storage. By using this model, the large lag times in data availability are eliminated, making analysts more effective and timely when dealing with this data set.

3 Related Work

Research into the area of abstraction for intrusion detection data sets has been ongoing for some time. The Common Intrusion Detection Framework (CIDF) is one such project [12]. The CIDF specifies protocols for IDS to interact with one another. The protocols are concerned with how individual tools interoperate, allowing for the detection of sophisticated attacks. A shortcoming of this project is that data management is not adequately addressed. When dealing with multiple data sources on large networks, data capture and management become very difficult. One primary goal of the work presented in this paper is to address this issue.

Additionally, a specification language, Common Intrusion Specification Language (CISL), is defined within the CIDF to allow the modeling of intrusions. This language shares many traits with the Intrusion Detection Message Exchange Format (IDMEF), a means of representing intrusion alerts using XML for interoperability [13]. These languages can be seen as complementary to the framework defined herein. Intrusion alerts represented in either form can be manipulated like any other data set within the system.

In another paper, Ning et al propose an extension to CISL to add query capabilities across CIDF modules [11]. This work is similar to the query language proposed in this paper. However, by the authors' admission, this extension is outside the scope of the original CIDF design. The query language defined herein is a fundamental aspect of the system, not an extension as in the case of the CIDF.

Performance and scalability are often afterthoughts in intrusion detection research. In many cases, projects are evaluated on a small, canned data set [1,5]. Real world analysis is typically a second step in the evaluation process [2]. Even if real-world analysis is performed, the size of the data sets used for testing are small relative to the data rates seen in large network installations.

4 Architecture

The framework architecture can be broken down into the following modules: the data provider module, the data server module, the dynamic index module, the static index module and the query module. These modules can reside on the same nodes or can be distributed across several nodes, allowing the system to scale. The data provider module is implemented by any application that wants to input data into the system. The data server module handles data capture from the data provider sends data items to the dynamic index module. The dynamic index module builds an index structure dynamically, allowing for data items to be inserted and retrieved in real-time. The static index module takes indices generated by one or more dynamic index modules and merges them into a static index structure. Lastly, the query module allows transparent data access to both dynamic and static indices. These modules form a powerful framework for collecting and managing intrusion detection and network traffic data sets. Multiple heterogeneous sensors can feed information into the system in real-time. The larger the cluster used to implement the framework, the more data that can be captured and analyzed. Both data capture and information retrieval are distributed, ensuring scalability.

Figure 1 illustrates a typical intrusion detection framework (IDF) configuration. An intrusion detection system (IDS) implementing the data provider module sends data, represented by the black arrow, to a node running a data server. This node is considered the master node, as it will be responsible for processing the data stream from the IDS. In this case the data server can send data to one or more subordinate nodes, labeled Sub_1 through Sub_n, in the cluster running the dynamic index module. When the operation is complete, the master node migrates the dynamic indices to a static structure using the static index module. Also note that all of the nodes have implemented the query module, allowing for distributed queries across the cluster.

Although only one IDS is shown in figure

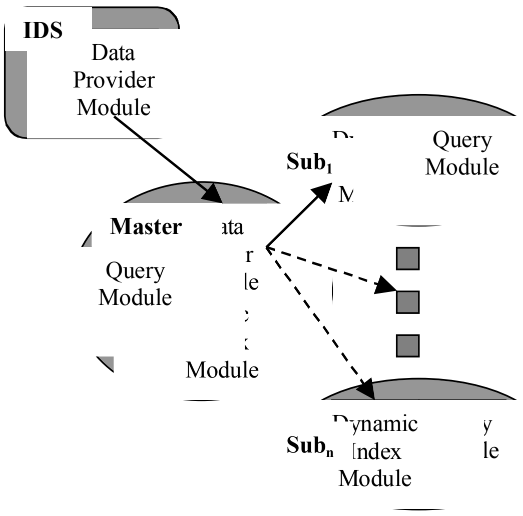

Figure 1. System architecture

1, numerous input streams can be handled by the data server at once, allowing the system to capture data from a variety of data sources. This allows the system to meet the needs of large, complex network installations. In many cases multiple sensors and data sources will be present and will need to be captured by the data server. Each data source or sensor must implement the data provider module to feed into the IDF. Scalability can be ensured by instantiating multiple data server modules, static index modules and dynamic index modules across a cluster.

The remainder of this section details the high-level concepts of the framework. Section 5 describes the implementation of the various modules in the prototype being developed at LANL.

4.1 Data provider module

A data provider is defined as any application that inputs data into the system. This could be a traditional IDS like Snort, sending IDMEF alerts, or a server sending system call traces. There are two conceptual requirements for a data provider: data registration and interface implementation.

Before a data provider can input data, it must register the type of data it will be sending to the system. Each data type must have its own unique description in the system's metadata. For instance, IDMEF alerts would have one entry. Linux system call traces would have an identifier, as would traces from a Windows machine. The metadata must be flexible enough to accommodate a wide range of data types. The requirements for the metadata vary for different data types. XML based data, like IDMEF, needs little more than the location of the DTD. Data such as system call traces requires a description of the fields that includes information like field lengths, field types and delimiters.

The second requirement, interface implementation, is rather straightforward. The data server, discussed in the following section, defines an application-layer header for data providers connecting to the data server. The data provider simply opens a socket connection to the data server, writes the application-layer header to the socket and starts sending data items. When the data provider is finished sending data, it simply closes the connection, signaling the data server that it has completed its operation.

One caveat is that the data server may deny the connection attempt from the data provider. If this happens, the data provider must either wait or attempt to connect to a different data server.

4.2 Data server module

The data server is the mechanism that provides scalable, high-throughput data capture. It allows applications implementing the data provider module to send data to the system without worrying about how or where the data will be stored. In a typical configuration, most of the nodes will be running a data server module so that one single node does not become a bottleneck, handling all data input.

The data server must listen for new connections from data providers. When a new connection is opened, the process must fork to allow additional data providers to connect. It is suggested that the data server redirect data providers to another node if the data server has become overloaded. If the number of currently connected data providers exceed a certain threshold additional data providers should be denied. An overloaded data server should suggest a different data server node for the data provider to interact with in its reply.

The data server processes an application layer header upon each connection from a data provider. This header includes the data type of the data being processed, the sensor that is sending the data and the date of the data set. This information must be recorded in the system metadata for use in other modules within the IDF.

The date information imposes the following restriction on the data provider: data streams must not span multiple days. Thus, the data provider must periodically (i.e. at midnight) close the data stream and restart it by reconnecting to the data server. This restriction is necessitated by the requirements of the query module. Although it may seem prohibitive in some cases, this type of data is time series data and can be easily segmented in this manner.

As data items are received from the data provider, they must be immediately sent to a local or remote dynamic index module. If remote index modules are involved, a load balancing function is applied to the data item to determine where to send the data item. Implementation of this algorithm is left up to the implementer, however it is suggested that system status and metadata be used to dynamically load balance the system.

4.3 Dynamic index module

After the data server processes data items, the dynamic index module must process them immediately to make data available in real-time. This module is responsible for storing the data items and making them accessible to the query module. The data items stored and indexed can be thought of as transient. That is, eventually the data and indices will be migrated to a permanent location by way of the static index module.

Upon entering the dynamic index module, a data item must be made accessible locally. Exactly how the data is stored is dependent on the needs of the implementation. Potential storage methods include relational databases, flat files or embedded databases. Next, the data item must be made accessible to the query module. How this is achieved is influenced by the means of storage. For instance, the query module would be easy to implement if the dynamic index module stored all its data items in a relational database; the dynamic index module would not need to perform any additional processing. However, if flat files were used as the storage medium, additional processing would be needed to provide the query module with enough information to retrieve individual data items.

4.4 Static index module

Data sets used in network traffic analysis and intrusion detection are append-only in nature. Once data is recorded, it will not change over time. This allows for various optimizations that are not possible when dealing with dynamic data sets. In the IDF, the static index module is where these optimizations can be implemented to leverage the properties of the data.

The static index module is used to migrate the data items collected by the dynamic index module to permanent storage. This process involves merging the data items from one or more dynamic index modules into one static result. This result contains two parts: the data and the access method. The data items are merged, if necessary, to create one master data set. After the data set is built, the static access methods must be generated. The static index module is an optional module in the IDF, as some access methods implemented in the dynamic index module may be sufficient for permanent data access. In this case, the static index module is not necessary and can be ignored.

To begin processing, the static index module on a designated master node must send a notification to any remote nodes in the cluster that have data items relating to the data set being processed. These nodes will become subordinate nodes in the static index building process. The subordinates must have some means of returning data to the master node, for example NFS or TCP/IP sockets.

Once all the data is received and the access methods are built, the static index module can migrate them to permanent storage. At this point, the module performs some clean up functions, including clearing out the dynamic index module's data sets, as they are no longer needed. Also, the module is required to update the system's metadata. The location, size and time information are updated in the metadata so that the query module, described below, can find the access method for the newly processed data.

Although this module is optional, it is useful in most situations. Even if a relational database was used in the dynamic index module as the storage mechanism, there is still much to gain from implementing a static index module. For example, the database used in the dynamic index module would be tuned for fast inserts. The static index module could migrate the records to a different database that was tuned for query performance.

4.5 Query module

The final and most important module in the IDF architecture is the query module. The purpose of this module is to search data sets using the access methods built and maintained and by the static and dynamic index modules. When a query is submitted to the query module, it divides the query into sub-queries if possible. These sub-queries can be executed sequentially or sent to query modules in remote nodes. In either case, all query results must be returned to the originating query module.

Queries submitted to the query module must be submitted as XML in adherence to the DTD shown in appendix A. This representation defines the IDF query language (IDF-QL). The schema described herein should be seen as a preliminary definition of the IDF-QL. Defining the query language is a driving factor in the current research direction of the IDF.

The IDF-QL is not tailored to any particular data set. However, it is apparent in appendix A that the attributes defined by the DTD are fields common to a variety of intrusion detection and network traffic data sets.

Date information must be provided in each query. At least one Date element or date attribute must be specified. This information is found in the system metadata and is recorded when a data provider connects to a data server.

To support the time attribute, time information must be indexed by an access method in both the static and dynamic index modules. This is the only attribute that is required of data sets within the IDF. If time information is not available within the data, the data provider module must modify the data stream in some way to include this information.

In addition to adhering to the schema defined in appendix A and the date/time requirements, some additional rules must be enforced by the query module when processing queries. Appendix B provides example queries to illustrate these rules. The first rule is that overlapping date (or time) elements (or attributes) cannot be nested in sub-queries. For example, the user cannot submit a query specifying a date range of March through April at the root Query element and then specify another date range in a nested query. Rule 1 is stated as follows:

Rule 1: *Nested queries adhere to the date/time ranges specified in the parent, unless the parent has no recursively specified date/time range*

The next set of rules deal with the handling of multiple data types within a single query. The optional Type element allows the user to specify what data types to return in the result set. Omission of the Type element implies that all data types across all sensors should be included in the result set. Also, like Rule 1, nested queries cannot have conflicting Type definitions. Rules 2 and 3 are stated as follows:

Rule 2: *Omission of the Type element implies that all data types and all sensors should be queried*

Rule 3: *Nested queries adhere to the type specification in the parent, unless the parent has no recursively specified type*

The next rule deals with mapping attributes in the Query element to the data sets being searched and returned in the result set. Not every data set has the search attributes for a given query. For instance, one data set may not have a MAC address field. When a query on MAC address is requested, that data set cannot be queried. Rule 4 is stated as follows:

Rule 4: *In order to be returned in a result set, a data set must contain all the search attributes in some form*

Finally, the result elements must follow the same property as defined for time and type: nested result types cannot conflict with the result type of the parent. Also, there is no default result type and some result type, or types, must cover the query. Rules 5 and 6 are stated as follows:

Rule 5: *Nested queries adhere to the result specification in the parent, unless the parent has no recursively specified result*

Rule 6: *A result element, or elements, must cover the query*

In summary, the query module is a critical component in the IDF and drives the design of many of the system's other modules. The static and dynamic index modules must provide access methods that support the functionality of the query module. As defined here, the query module provides analysts with a powerful means of accessing heterogeneous data sets with an intuitive, yet powerful, query language. Further definition of the IDF-QL is ongoing and will undoubtedly have impact on the system architecture. Refer to appendix B for example queries.

5 Implementation

As of the writing of this paper, all of the modules described in section 4 have been implemented to provide data capture and data access to the NIDS data set. As shown in Figure 1 above, the system architecture is a distributed environment, however this is not a requirement. Section 5.3 explains how the system functions in a single-node environment. This approach allows the framework to scale from one to many nodes, depending on the user's needs. In most cases, the distributed approach is preferable.

Three entities are represented in Figure 1: the data provider, the master node and the subordinate nodes. The data provider, described in section 5.1, connects to the data server on the master node. The master distributes the load amongst the subordinates. Any tool generating intrusion detection data can become a data provider. Potential data to be collected could include IDMEF alerts, web server logs, system call traces or network connection summary data like LFAP or Netflow. Each tool is associated with system-wide metadata describing the data format. Some of this information includes field lengths, delimiters and data types (i.e. long, integer, string). The metadata allows other entities in the system to properly parse and manipulate incoming data.

The master node performs much of the critical processing of the data stream. As data items are read from the data provider, the master distributes them to the subordinate nodes, which are responsible for maintaining the dynamic indices. Currently round robin scheduling has proven sufficient for load balancing the system. Whenever a data provider closes its connection to the master without error, the master begins to migrate the data to static index structures. This process is explained in section 5.4. The static indices are currently implemented as b+ trees with fully filled pages. The keys are user-defined field contents, while the data values are file offsets.

As subordinate nodes receive data items from the master, they must parse the data item and update the dynamic indices. The current framework prototype utilizes Berkeley DB to serve as the dynamic data store. This allows for high data capture throughput and fast, flexible data access as shown in section 5.1. Because the indices are updated in real-time, the current query module implementation can retrieve data as soon as the subordinate processes it. The choice of what fields to index is left to the user and is defined in the system metadata and configuration parameters.

At this point it is worth commenting on the use of static b+ trees and Berkeley DB as the data access methods. Clearly, the use of a relational database provides much more flexibility than the approach used in this case. However, as shown in section 6.1, the throughput suffers considerably when a relational database is used as the data store. This is a critical factor when dealing with high-speed networks with high data volumes. Also, section 6.2 shows that storage overhead is higher compared to the approach implemented. The desire to avoid data redundancy and to keep the data in its original format was a crucial design factor in the framework as well. Often, custom tools have been written to access the data in its raw state. The user might then be forced to choose between discontinuing the use of these tools and having redundant data. In most cases, having two copies of a terabyte-sized data set is impractical and expensive.

5.1 Data provider

There are currently two implementations of the data provider module, both designed to send NIDS data to a data server. The first implementation is a simple, 136 line Perl script which reads in a NIDS data file, opens a connection to a data server and sends the file contents over the socket. The primary purpose of this script is to illustrate how easy it is to implement a data provider module. Aside from some information in the IDF metadata, this is the only aspect of the implementation that the data provider need be concerned with.

The second implementation has been tested in a development and is built into the NIDS application. As soon as a record is created by the NIDS, it is written to a socket. This data provider module, written in C, takes fewer than 100 lines of code. When deployed in production, NIDS records will be available in real-time allowing for accurate and timely analysis of network activity.

5.2 Data server

In the current IDF configuration, all of the nodes run a data server so that one single node does not become a bottleneck. Each data server listens for connections and spawns a new process for each successful connection. At this point, the data server processes an application layer header that contains information about the operation type, either an insert or a dump request, and the data type, a global identifier tying the data type to information in the system metadata.

After processing the first portion of the header, a function specific to the data type and operation is called to perform the majority of the processing. The purpose of the dump request is discussed in sections 5.5 and 5.6 below. Insert operations are handled by application-specific functions, one for each data type. Currently, only NIDS data is supported, however it is easy to add support for other data types with similar properties such as router logs or system logs.

Before data can be processed, the application-specific function must process any additional header information. Seven additional parameters are processed when performing insert operations on NIDS data: three integers that tell how to distribute the processing and four time-related integer parameters. The state of the first three integers will result in one of the following three cases:

single-node server, master server or subordinate server. Sections 5.3, 5.4 and 5.5 deal with these three cases respectively. The next three parameters are date information for the incoming data: month, day and year. The final parameter is the "run" number, which allows for multiple builds of data from the same day. This is used for high volume situations when the user wants to cap the number of data items in a particular index structure. For example, the user could decide that once 10 million data items are sent that a new run should be started. The application implementing the data provider simply closes the socket and opens a new one after incrementing the run number.

The current data server prototype is written in Java and supports TCP or UDP sockets, optionally using SSL for secure data transmission. These decisions are left largely up to the developers based on the needs of their environment. It is recommended that SSL be used within a secure network to ensure information safety and system reliability.

5.3 Single node server

In some cases, it is desirable to only use a single node to handle data capture. For instance, a low-bandwidth network feed might only need a single server to manage incoming data. The single node server performs four primary functions: reading data items from the client application, parsing the data items, inserting into the temporary data store and migrating to the permanent data store. The first three functions commence whenever a client application connects to the data server and continue as long as the connection remains open. Migration takes place when the connection is closed, or at the data server's discretion.

Upon client connection, the data server must prepare the various data stores needed to build the dynamic indices. First the server must check to see if a database environment has already been instantiated. Berkeley DB was chosen for the current implementation because it is open source, embedded and provides high throughput. The database environment in Berkeley DB has several functions including buffer pool management, transaction logging and disaster recovery. If no database environment exists, one must be created. Once an environment is instantiated, the server can create the new dynamic indices. Each field in the data set being indexed needs its own database. The field values become the database keys in the b+ tree while the data values become the offsets of a temporary data file. This data file is appended to as new data items are received. A delimiter, in the case of NIDS records a new line character, signals the end of a data item. Finally, a parser must be instantiated based on the data type metadata.

Once the above steps are completed, the node can begin performing inserts into the dynamic indices. As data is received, it is sent to the parser, which returns the fields of the data item as an array of strings. The data item is appended to the temporary data file at this point as well. Next, each field being indexed is inserted into its corresponding database along with the current file offset. Once these operations complete, the data item can be discarded and a new one can be processed. Also, at this point the data item is available to be retrieved through the query module.

When the data provider closes the connection, the node can begin the migration process, which will result in the generation of static b+ trees, one for each dynamic index. In the case of the single-node server, this process is very simple. For each dynamic index, the node simply traverses the database in ascending order and writes the key/data pairs to a temporary file that will be used to build the static b+ tree.

Once this operation is completed, the node calls the index builder application and waits for it to complete. The index builder is written in C++ and implements b+ trees [14] in flat files. Upon successful completion, the node can clear out the dynamic indices and any temporary files that are no longer needed.

5.4 Master server

The master server mode of operation is similar to the single-node case with the addition of distributed data processing on remote subordinate nodes. Generally speaking, the master reads data items from the provider and sends the record to a subordinate. Currently, round robin scheduling has proven sufficient as the machines are essentially identical in performance and perform the same set of tasks. When the connection from the data provider closes, the master gathers the results from the various nodes, merges them together and builds a static b+ tree index.

The initial steps for the master node are somewhat different from the single-node case. Depending on the application layer connection header, the master may or may not record data items. If not, the master becomes a "splitter node" that simply reads the data in and round robins data to the subordinates. This mode is typically used when dealing with very high data rates. In practice, the splitter node has shown to out-perform a configuration with the master recording data items.

Whether or not the master is acting as a splitter, it needs to open a connection to a data server on each of the subordinate nodes. If the master node is also recording data items then it must follow the same initialization steps as a single-node server. At this point the server can start receiving data from the data provider application.

The scheduling algorithm, in this case round robin, determines the next step taken by the master server for a given record. If the scheduler maps a record to the master node it must update the dynamic indices and write the data item to disk, as in the single-server case. Otherwise, the data item is sent to the corresponding node's dynamic index module via the node's data server.

Data items are processed until the data provider closes its connection to the master. Then, the master closes its open connections to the subordinates. At this point, the master enters the merge phase, which is described in section 5.6. The process involves reopening connections to the subordinates and sending a "dump request" to each. The dump results are then merged to yield one data set. After the merge successfully completes, the master behaves just as a single-node server and builds the static b+ tree indices.

5.5 Subordinate

Each subordinate node has two forms of operation: data capture mode and dump mode. While in data capture mode, the subordinate receives data, parses it and updates the dynamic indices. This is no different than how the single-node server case handles data. Dump mode becomes important when the master is ready to merge the data set and build the static indices. The master reopens connections to the subordinates, this time specifying in the header that a dump of the dynamic indices should be created. The subordinate writes the contents of each dynamic index to a temporary file accessible to the master node over NFS. The details of this process are described in the following section.

5.6 Index building

When a data provider closes a connection to the data server, dynamic Berkeley DB indices can be migrated to static b+ tree data structures. This operation potentially requires three steps: merging data from subordinates, running the index builder application and final migration. Merging requires the raw data and the dynamic indices of the subordinates, if any, to be combined on one machine. This phase prepares the data for the index builder application. Optionally, the raw data and static indices can be migrated to a final destination, an NFS server for instance, to allow access to the data and indices from multiple machines.

At the beginning of the merge phase, each subordinate node, and optionally the master node, holds a portion of the data set. First, the master sends a dump request to each subordinate. The subordinates then dump their dynamic indices in parallel. In the prototype implementation, these dumps are placed in locally exported NFS partitions to which the master has read access. The dump operation writes the key/data pairs to a flat file, one pair per line, while traversing the dynamic index in ascending key order.

After dumping the dynamic indices, the index builder application is executed to generate the b+ trees. The input to the application is a file containing the processing parameters, which is generated by the data server prior to calling the index builder. The types of information in the input file include date information, the data file path, the type of data, fields to index and node information.

Given this information, the index builder takes the temporary data files on the subordinate nodes and concatenates them to create one large data file. At this point, the index dumps still need to be merged from the subordinate nodes. For each index being built, the corresponding dumps are merged into a single file. The file offsets in the dump files need to be adjusted to reflect the fact that the records are now part of a single, larger file.

The resulting file contains one line for each unique key value. A delimiter, if needed, follows the key value and then the file offsets of all the data items containing that key. This "master dump" file will be stored permanently and is used by the query module to retrieve data items.

5.7 Query module implementation

At this time, the query module implementation supports the features of the IDF-QL as defined in section 4.5. A query processor application handles query requests from clients and returns the results. It also performs validation checking and enforces the rules defined by the IDF-QL. Once a query has been validated, the query processor translates the query into a call to an external search application that feeds results back to the query processor and ultimately to the client. The search application is designed to traverse the b+ tree data structures and the Berkeley DB-based dynamic indices transparently and return matching results. It performs joins on search criteria using a traditional sort-merge join [14] and has a simple but effective caching mechanism. The cache stores the results of elementary queries and leverages these stored results to increase query execution time. Information about what results are cached is stored in the system metadata. Data in the cache is periodically flushed to avoid filling up a node's local storage. The cache is very effective when a query is repeated in total, or with minor modification. Initial performance results of the query module implementation, as shown in section 6.3, are encouraging.

6 Benchmarking

The primary motivation of this section is to provide evidence to support the claim that a relational database is not an adequate solution to the problems inherent to large network traffic data sets. Section 6.1 discusses system throughput, first when using a relational database as the backend data store a then when using Berkeley DB-based indexing techniques. Scalability is also discussed therein. In section 6.2, the overhead of storing network traffic data is evaluated. Naturally, any sort of table structure or index scheme imposes additional overhead above what the raw data requires. This section illustrates the storage savings when comparing static indexing techniques to relational databases. Lastly, section 6.3 provides initial query performance results, comparing the static b+ tree data structure to MySQL for a small set of queries.

6.1 Throughput and scalability

As discussed in Section 4.2, data capture is a critical element in the system architecture. A system attempting to capture large network traffic data sets must be able to keep up with the high volume and potentially bursty nature of the data. Therefore, throughput becomes an important metric when evaluating these systems. Also, throughput is closely tied to scalability. A scalable system is necessary to meet the needs of a dynamic and rapidly growing network.

NIDS data from two days was used to evaluate the throughput of the system prototype. The first data set was from April 1, 2002, containing approximately seven million records. The second data set was from September 14, 2003 and contained approximately 20 million records. The large difference in the number of records can be attributed to an increase in network activity coinciding with the Microsoft DCOM vulnerability. These data sets were chosen because they illustrate how rapidly the size of the data sets grow. Due to the increase in port 135 and 445 scans, the number of NIDS records tripled over a one-week period. Until the number of DCOM scans increased so dramatically in late August 2003, the April 1 dataset was very representative of a typical day's worth of NIDS data. Any system hoping to perform data capture of network traffic data must be able to handle rapid increases in data volume of this sort.

The primary design decision that impacts throughput is the backend storage mechanism. The most obvious solution is to insert records into a relational database. MySQL (version 3.23.53) was chosen in this case because it is widely used in intrusion detection systems like Snort and thus many

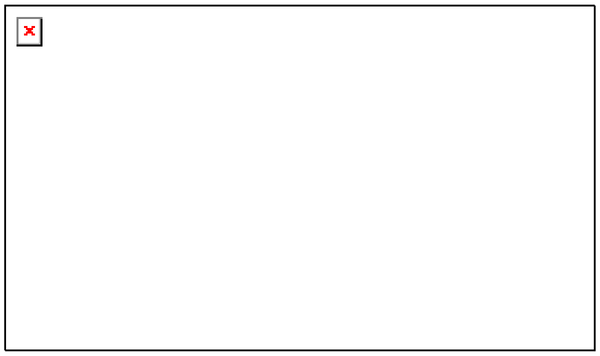
Figure 3. System throughput

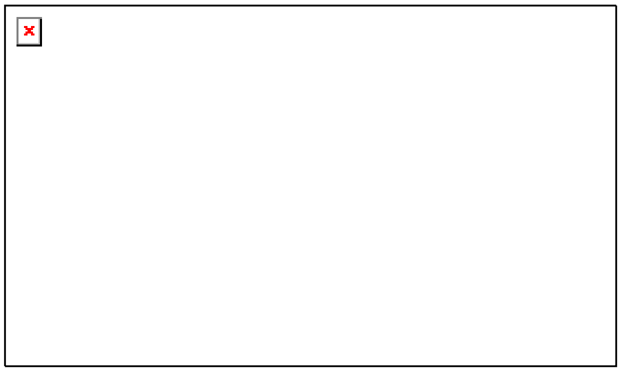
Figure 2. Backend data store performance

intrusion detection analysts have experience in its operation. For this evaluation, records were inserted one by one into an empty table. No optimizations were made to the database environment. Two dual Pentium III 1 GHz computers with 1 GB RAM and gigabit Ethernet were used for the test. One machine implemented the data provider module while the other implemented the data server and dynamic index modules. Figure 2 illustrates the results of this benchmark.

Although the throughput rate was acceptable for the 2002 data, the throughput fell 46% on the 2003 data to an average of 135 records per second. At this rate, it took 1.6 days to load the data set into MySQL. By comparison, the same test using Berkeley DB (version 4.1.24) as the backend as described in section 4 yielded superior throughput while only showing a 24% degradation in performance on the larger data set. By contrast this data set took only 13 hours to load using Berkeley DB.

Although using MySQL's bulk data load capabilities would yield better results, this would not satisfy the real-time requirement of the system. Also, it is certain that database tuning would result in better throughput for MySQL, although similar optimizations could also be applied to Berkeley DB. The key point made here is that MySQL's additional complexity and overhead results in reduced throughput when compared to using Berkeley DB as the backend data store.

Figure 3 illustrates scalability, another key element in the system design. To deal with rapidly growing data sets, the system must be able to scale. For this benchmark, three configurations were evaluated: single-node, three-node and five-node. In each case there was one master node implementing the data server. In the one-node setup, the master handled all the processing, implementing the dynamic index module as well. In the three and five node setups, each subordinate node implemented the dynamic index module to allow for distributed processing of the incoming data. As shown, the system scaled well over the 2002 data set. Resource and time constraints limited the ability to test the scalability of larger environments. The five-node environment's throughput has proven to be more than sufficient for current data rates.

One additional point about Figure 3 requires clarification. The throughput increased 138% from the single-node environment to the three-node environment. The throughput increase from the three-node case to the five-node case was 180%. The disparity here stems from the fact that the three-node environment had only one more dynamic index module implemented than the single node case - one on the master for the single-node configuration and one on each of the subordinates for the three-node setup. The five-node environment had two more indexing modules than the three-node setup, accounting for the 42% difference in performance. Thus the initial results of the scalability benchmarks show that adding one dynamic indexing node results in around 40% increase in throughput. This result is encouraging, although additional research is needed to verify this claim on multiple larger data sets and larger clusters.

6.2 Storage overhead

The desire to keep storage overhead at a minimum was a key driver in the IDF design and implementation. Two long-term storage methods were examined: relational databases, in this case MySQL, and static b+ trees. Both approaches have advantages and disadvantages. Relational databases provide the user with a great deal of power in manipulating the data. The drawback is that the overhead is high and it introduces data redundancy. One of the design goals was to keep only one copy of each data set online. Storing each record in a relational database and storing the raw data essentially doubles the size of the data set. To avoid this, static b+ trees were evaluated. The advantage of this

	Time (sec)			
Query	B+ Tree	MySQL	% Diff	# Results
1	7.84	0.45	5.69%	18
2	0.37	0.03	6.94%	184
3	0.36	1.60	448.37%	758
4	98.16	1,012.94	1,031.93%	820,410
5	136.53	1,831.13	1,341.19%	369,186

Table 2. Query performance

approach is that the raw data itself is indexed, eliminating the need for an additional copy of the data. The disadvantage is that the analyst no longer can leverage the power of a relational database. However, given the nature of the types of queries used by analysts, the b+ tree solution provides enough power to compute the majority of these queries, while maintaining a low overhead.

For the purposes of this benchmark, three cases were examined: a relational database with raw data, a relational database without raw data and static b+ trees with raw data. MySQL 3.23.53 was used on Mac OS X version 10.2.7. For both the b+ tree and MySQL databases, four fields were indexed: destination address, source address, destination port and start timestamp. Table 1 shows the results of the benchmark.

Storage Method	Size (MB)	Ratio
Raw	858.41	100.00%
Raw + Index	1124.52	131.0%
MySQL	1523.60	177.49%
Raw + MySQL	2382.01	277.49%

Table 1. Storage overhead

As is evident in Table 1, using static b+ trees requires around 45% less overhead than using MySQL as the access method without storing the raw data. However, since it necessary to save the data in many cases, the overhead becomes substantial. These issues led to the adoption of the b+ tree solution in the current IDF implementation, although other solutions could be implemented to meet the needs of the user.

6.3 Query performance

A set of five different queries were run five consecutive times on matching dual Xeon servers with 2 GB of RAM to evaluate the performance of the b+ tree index structure. MySQL version 4.0.18 was used for comparison in this test. Table 2 illustrates the averaged results from each query.

The data set used was a single high volume date from September of 2003. Since there are four NIDS data files generated per day, four MySQL databases were built to correspond with this arrangement. Each database was indexed on destination port and source address.

For brevity, the actual query syntax used to generate the above results is not included. Query 1 returned results matching on a source address and destination port. Query 2 returned results from the same destination port as query 1 but with no restriction on source address. Query 3 returned results matching on a small range of destination ports. Query 4 returned results matching a highly active destination port. Query 5 returned results matching a larger port range than Query 3.

As is evident in table 2, the static b+ tree structure performed well for cases generating a large number of results. The results from queries 1 and 2 were substantially better when using MySQL, as the caching mechanism helped consecutive runs perform very well. Even so, the total time to return these queries was low when using either system.

These initial results support the decision to use the b+ tree data structure as the underlying data retrieval mechanism. Performance is very good when dealing with large volumes of data, as was the initial design goal of the system. It is also reassuring that queries with small result sets also return quickly, although not as fast as when using MySQL in some cases.

7 Future Work

The most pressing task in the development of the IDF at LANL is the inclusion of other data types. Although the NIDS data set is an interesting data source, it does not meet the needs of all situations. Additional data sets such as router logs and IDS alerts need to be integrated to provide analysts with a more comprehensive view of network activity. Adding new data types may also lead to creating new access methods other than the b+ tree solution.

The query module is an area of research that will receive a large amount of attention. The definition of the IDF-QL as defined here is a preliminary definition. The current implementation provides adequate functionality for many task related to network traffic analysis and intrusion detection but requires additional research to realize its potential. Once the query language is complete, attention can be shifted to focus on distributed query processing and optimization.

In addition, robustness and survivability must be addressed. As with any distributed system, there are many potential points of failure. These areas must be better understood so that the system can survive failures at various levels. Currently, node failures are addressed by simply ignoring the node until it becomes available. A more robust fail-over system needs to be implemented so that data capture can continue unhindered when a node fails.

Lastly, an exhaustive benchmark of the system must be made. Areas to address include throughput, scalability, robustness and query performance. The results presented here are encouraging but not comprehensive. Additional research must focus on this aspect of the system.

8 Conclusions

In this paper, a framework for the collection and management of intrusion detection data sets is proposed and initial results are reported. The design of the system was driven by the network security needs inherent in high-volume network installations. Managing the data sets associated with network traffic and intrusion detection becomes very inefficient using current techniques. Therefore, a uniform mechanism for capture and retrieval of this data is essential to enable security analysts to effectively perform their duties. The initial results of the IDF implementation being tested at LANL have been encouraging. High throughput data capture and retrieval has been shown using the high volume NIDS data set as a test case. Furthermore, the static b+ tree indexing methods have shown promise in minimizing storage overhead while still providing analysts with a powerful information retrieval tool. Additional work in data set integration, query capability and robustness need to be performed before the IDF can be a viable solution for network security at large installations.

References

[1] C. Taylor and, J. Alves-Foss, "NATE - Network Analysis of Anomalous Traffic Events, A low-cost approach", *Proceedings of the 2001 workshop on New security paradigms*, ACM Press, New York, NY, 2002, pp. 89-96.

[2] C. Taylor and, J. Alves-Foss, "An empirical analysis of NATE: Network Analysis of Anomalous Traffic Events", *Proceedings of the 2002 workshop on New security paradigms*, ACM Press, New York, NY, 2002, pp. 18-26.

[3] T. Lane and C. E. Brodley, "Temporal Sequence Learning and Data Reduction for Anomaly Detection", *Proceeding of the 5th ACM conference on Computer and communications security*, ACM Press, New York, NY, 1998, pp. 295-331.

[4] F. Cuppens, A. Miege, "Alert Correlation in a Cooperative Intrusion Detection Framework", *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 187-200.

[5] W. Lee and S. J. Stofolo, "A framework for constructing features and models for intrusion detection systems", *ACM Transactions on Information and Systems Security*, Volume 3, Issue 4, ACM Press, New York, NY, 2000, pp. 227-261.

[6] X. Yang, J. Shen and Q. Liu. "A Novel Clustering Algorithm Based on Weighted Support and its Applications", *Machine Learning and Cybernetics*, Volume 1, 2002, pp. 95-100.

[7] L. Teo, Y. Zheng, and G. Ahn, "Intrusion detection force: an infrastructure for internet-scale intrusion detection", *Proceedings of the First IEEE International Conference on Information Assurance*, 2003, pp. 73-86.

[8] T. Bass, "Intrusion detection systems and multisensor data fusion", *Communications of the ACM*, Vol 43, Issue 4, ACM Press, New York, NY, 2000, pp. 99-105.

[9] F. Cuppens, "Managing Alerts in a Multi-Intrusion Detection Environment", *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001, pp. 22-31.

[10] P. Ning, S. Jajodia and X. S. Wang, "Abstraction-based intrusion detection in distributed environments", *ACM Transactions on Information and System Security*, Volume 4, No. 4, ACM Press, New York, NY, 2001, pp. 407-452.

[11] P. Ning, X. S. Wand and S. Jajodia, "A query facility for common intrusion detection framework", *Proceedings of the 23rd National Information Systems Security Conference*, 2000, pp. 317-328.

[12] M. Reilly and M. Stillman, "Open infrastructure for scalable intrusion detection", *IEEE Information Technology Conference,* 1998, pp. 129-133.

[13] D. Curry and H. Debar, "Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition", http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-10.txt, 2003.
[14] R. Ramakrishnan and J.D. Ice (ed), *Database Management Systems*, The McGraw-Hill Companies, Inc., USA, 1997.

Appendix A: IDF-QL DTD

A preliminary document type definition (DTD) for IDF-QL is provided below. The IDF-QL is currently being enhanced and thus this DTD should not be seen as a definitive specification of the language.

```
<?xml version="1.0"?>
<!ELEMENT Query (Query*, TCPResult?,
DBResult?)>
<!ELEMENT SocketResult EMPTY>
<!ELEMENT DBResult EMPTY>
<!ATTLIST Query srcip CDATA>
<!ATTLIST Query srcmac CDATA>
<!ATTLIST Query srcport CDATA>
<!ATTLIST Query dstip CDATA>
<!ATTLIST Query dstmac CDATA>
<!ATTLIST Query dstport CDATA>
<!ATTLIST Query date CDATA>
<!ATTLIST Query time CDATA>
<!ATTLIST Query dataset CDATA>
<!ATTLIST Query sensor CDATA>
<!ATTLIST SocketResult host CDATA>
<!ATTLIST SocketResult port CDATA>
<!ATTLIST SocketResult type CDATA>
<!ATTLIST DBResult host CDATA>
<!ATTLIST DBResult port CDATA>
<!ATTLIST DBResult db CDATA>
<!ATTLIST DBResult database CDATA>
<!ATTLIST DBResult table CDATA>
```

Appendix B: IDF-QL examples

The following examples illustrate several correct and incorrect queries specified in IDF-QL.

```
<Query srcip="1.2.3.4" dstip="5.6.7.8-
9.8.7.6" dstport = "1-1024" date="1/1/2002-
12/31/2002">
   <SocketResult type="tcp" host="3.4.5.6"
port="10000"/>
</Query>
```

```
<Query>
   <Query srcip="5.6.7.8" date="9/1/2003-
9/10/2003">
       <DatabaseResult host="3.4.5.6"
port="3306" db="MySQL"
database="temp" table="results1">
   <Query/>
   <Query srcip="9.8.7.6" date="8/1/2003-
9/15/2003">
       <DatabaseResult host="3.4.5.6"
port="3306" db="MySQL"
database="temp" table="results2">
   </Query>
</Query>
```

The two queries above adhere to the DTD specified in appendix A and do not violate any of the rules from section 4.5. The first query performs a three-way join on source IP address, destination IP address and destination port for all dates in the year 2002. The results of this query will be sent to a TCP/IP socket on the specified host and port. The second query has two sub-queries, which search for two different IP addresses over different date ranges. The results from each sub-query are placed in a MySQL database on the same host in different tables.

<Query ***date="8/1/2003-9/15/2003"****>*
   <SocketResult type="tcp" host="3.4.5.6" port="10000"/>
   <Query srcip="5.6.7.8" ***date="9/1/2003-9/10/2003"****/>*
   <Query srcip="9.8.7.6" ***date="8/1/2003-9/15/2003"****/>*
</Query>

The above query violates rule 1 as described in section 4.5. The date ranges in bold are conflicting in that the parent query's range is not equal to the sub-query's range.

<Query ***type="1"****>*
   <SocketResult type="tcp" host="3.4.5.6" port="10000"/>
   <Query srcip="5.6.7.8" date="9/1/2003-9/10/2003"/>
   <Query ***type="2"*** *srcip="9.8.7.6" date="8/1/2003-9/15/2003"/>*
</Query>

The above query violates rule 3 as the types in bold are conflicting.

```
<Query>
    <SocketResult type="tcp" host="3.4.5.6"
port="10000"/>
    <Query srcip="5.6.7.8" date="9/1/2003-
9/10/2003"/>
    <Query srcip="9.8.7.6" date="8/1/2003-
9/15/2003">
        <SocketResult type="tcp"
host="3.4.5.6" port="9000"/>
    </Query>
</Query>
```

This query is in violation of rule 5, as the result element of the sub-query is not consistent with the result element of the parent in that they have different ports specified.

```
<Query>
    <Query srcip="5.6.7.8" date="9/1/2003-
9/10/2003"/>
    <Query srcip="9.8.7.6" date="8/1/2003-
9/15/2003">
        <SocketResult type="tcp"
host="3.4.5.6" port="10000"/>
    </Query>
</Query>
```

This query violates rule 6: the result element does not cover the query. Only the second sub-query has an associated result element. One must be specified for the first sub-query as well.