Advanced Indexing Techniques for Wide-Area Network Monitoring

Abstract

Detecting and unraveling incipient coordinated attacks on Internet resources requires a distributed network monitoring infrastructure. Such an infrastructure will have two logically distinct elements: distributed monitors that continuously collect packet and flow-level information, and a distributed query system that allows network operators to efficiently correlate information from different monitors in order to detect anomalous traffic patterns. In this paper, we discuss the design of MIND, a distributed index management system that supports the creation and querying of multiple distributed indices. MIND essentially builds a hypercube overlay on which two data records belonging to an index and near each other in the attribute space (the multi-dimensional space defined by the attributes of the data record) are likely to be stored at the same node. We validate our design decisions using traffic traces from two large backbone networks, then examine the performance of a MIND prototype on more than 100 PlanetLab machines. Our experiments use backbone traffic data sets and reveal that MIND can detect suspiciously large flows or DoS or scans in about a second, and that MIND exhibits perfect recall even with the failure of up to 15% of the overlay nodes.

1 Introduction

The advent of distributed attacks and self-propagating worms has spurred researchers and industry alike to explore innovative intrusion or anomaly detection systems. The state of the art in operational anomaly detection is to insert a physical system in front of routers, mostly on edge links, to detect anomalies, intrusions, and exploits using pre-defined rules [14, 20] or directly from traffic observations [13, 18, 11].

Such systems have two important drawbacks. First, they have no means to correlate their observations with other traffic monitors in order to reduce the likelihood of false positives. Second, they often do not support historical analysis of network traffic which allows reconstruction of past network events, identification of all compromised hosts or detection of vulnerabilities in the network infrastructure that could lead to future intrusions and attacks.

The next generation of anomaly detection system that overcomes these drawbacks will observe and maintain a detailed record of traffic data (e.g., packet-level traces) from most (if not all) links of a network. Such a widespread monitoring infrastructure would collect and store a massive amount of information that can then be monitored for anomalies or examined for post-mortem event reconstruction. The size of this dataset has two major consequences on the design of this new class of anomaly detection systems: (i) it is not feasible to “move around” or transfer the datasets to a centralized location, and (ii) fast and effective search methods are needed to navigate through the datasets.

A plausible way to architect these systems is as two logically separable components: a set of distributed traffic monitors deployed on most network links, and a querying system that enables fast correlation of traffic data. The traffic monitors will collect and store network traces, and they will also generate traffic summaries in the form of flow records or suitably aggregated and filtered versions thereof. The querying system will allow users (or other monitoring systems) to efficiently query these traffic summaries. The results of these queries will pinpoint which monitors contain relevant traces that may be further analyzed to confirm a potential intrusion or identify all compromised hosts to be quarantined, for example. Thus, such a system lets users efficiently drill down to important data locations. The querying system will likely support several kinds of queries, ranging from exact match to flexible pattern matching on flow records.

Our focus is on supporting one kind of query that is crucial to network monitoring: multi-dimensional range queries. Many queries that attempt to correlate traffic summaries are naturally expressed as multi-dimensional range queries. For example, a query of the form "was there a flow of size greater than 100MB to prefix X in time interval T?" can be used to monitor traffic volumes to a collection of customers. Similarly, a query of the form "what is the count of distinct sources sending small flows to port 3306 to destination prefix Y?" can be used to detect suspicious port scanning activity.

In this paper, we explore (Section 3) the design and implementation of MIND, a distributed system that supports the creation and querying of multi-dimensional indices. A distributed design of the querying system is attractive for reasons of scaling and robustness. A carefully designed distributed system can ensure higher availability, fewer bottlenecks, and present a lower profile to attackers. All of these traits are particularly important for network monitoring systems that need to be responsive precisely when the network is most challenged by correlated failure, or by large traffic shifts due to routing anomalies or flash crowds.

MIND consists of a collection of network nodes that forms a hypercube overlay; these nodes are logically distinct from, but could be co-located with, network monitors. Traffic summaries expressed as multi-attribute data records and generated at a network monitor can be inserted into one or more MIND indices. MIND routes these tuples to nodes such that tuples near each other in the attribute space are likely to be stored at the same node, making multi-dimensional range searches efficient. In addition, MIND contains a novel balancing technique that avoids storage hotspots arising from skewed data distributions.

Care must be taken in using a system like MIND for network monitoring. Clearly, it is infeasible to insert all flow records from each network monitor into MIND; such an approach could incur significant traffic overhead and could impact network performance. Rather, we see MIND as being used in much the same way database administrators build cen-
entralized indices. A network administrator performs careful off-line analysis to decide the attributes to be indexed, and the granularity of traffic summaries to be inserted into MIND. This database design analysis is based on the trade-off between the cost of building the index, and the expected frequency of querying the system. We analyze the feasibility of building indices that aim to keep track and enable fast access to suspicious flows (e.g., contributing an abnormally large volume of traffic, involved in port scan activity, etc.). However, MIND is not limited to this type of queries but is a generic platform that enables any kind of query index tailored to the needs of the network operators.

We have implemented a full-featured MIND prototype and have extensively experimented on PlanetLab using traffic flows records from two large academic backbones: Abilene [1] and GÉANT [2]. In one of these experiments, we carefully constructed a MIND overlay containing 34 nodes that matches the geographical and topological distribution of Abilene and GÉANT backbone routers and inserted over 9 million flow records into the system. In another, we evaluated the performance and robustness of MIND at a scale of more than 100 nodes. In a third, we examined whether MIND queries captured (after the fact) anomalies detected by an independently-designed off-line trace analysis algorithm [3].

Our results reveal that MIND can support median insertion and query latencies under 1 second, and can provide, with each data item being replicated once, accurate replies to queries even when 15% of the nodes fail. MIND exhibited perfect recall on all the anomalies we searched for, with average response times on the order of a second. Across all our experiments, we also observed the performance of paths that we can attribute to the experimental nature of the PlanetLab testbed.

These results suggest that MIND can efficiently support post-mortem analysis of network events. If deployed within a backbone ISP on a dedicated infrastructure, we believe MIND can be used as a component of an on-line anomaly detection system as well.

2 Motivation, Architectural and Design Considerations

We have said that the next generation of traffic monitoring systems will be architected as two separable components: a collection of distributed traffic monitors deployed on most network links, and a querying system that enables fast correlation of traffic data.

In this architecture, each traffic monitor collects and stores the entire sequence of packets observed at each link attached to the monitor. In addition, a traffic monitor generates flow records in real-time. A flow record typically counts the volume of traffic over a specific time window of a traffic aggregate. Traffic aggregates can be flexibly defined in terms of nodes, applications, or collections thereof. Thus, a traffic aggregate defined by the destination \( http://www.foo.com/ \) and a port number (say, 80) measures the volume of Web traffic to that destination. Another example of a traffic aggregate is the Web requests sent from a set of nodes to a single server. Often, given the way node addresses are assigned, “interesting” sets of nodes are usually represented by IP address prefixes (e.g., 192.168.32/20).

The design of traffic monitors represents a significant ongoing research challenge [4,5]. However, the design of sophisticated systems to query, search and detect correlations in flow records has yet to receive the level of attention it deserves. In such a system, the results from an appropriately designed query (or, more generally, from a sequence of searches) can, for example, narrow down the sources of a distributed attack to a subset of monitors. Having determined this, a network operator or a software system can drill-down into those monitors and examine detailed packet traces stored therein, in order to robustly detect anomalies.

In this paper, we design, implement and evaluate MIND, a querying system that provides a robust and efficient way to correlate flow records from a potentially large network of traffic monitors. MIND focuses on supporting multi-dimensional range queries\(^1\), which are perfectly suited for network traffic monitoring. Network traffic aggregates are naturally described by multiple attributes: source and destination addresses and port numbers, size, duration etc. Network monitoring in general, and anomaly detection in particular, requires the ability to flexibly examine different traffic “volumes”, and multi-dimensional range queries are ideally suited for this.

For example, consider the problem of detecting unusually large-volume point-to-point flows destined to a customer (usually identified by an IP prefix). These alphasflows might be indicative of on-going attacks [6], but might also be used to estimate network performance [7,8]. Potential alphasflows can be detected using a query of the form: \( \text{Find all flows destined for } D \text{ (and/or sourced from } S) \text{ that have carried at least } O \text{ octets}. \) This query defines a 5-dimensional data space: \( (\text{destination}, \text{source}, \text{size}, \text{node}, \text{timestamp}) \).

A second example of multi-dimensional range query is the detection of network scanning activity that often precedes an attack or exploit. Potential network scans can be detected using a query of the form: \( \text{Find all the sources that attempted to connect to more than } P \text{ hosts in destination prefix(s) } D \). Note that destination \( D \) (as well as source \( S \) if used) is in fact an IP prefix and so represents continuous ranges of IP addresses. This query defines a 5-dimensional data space: \( (\text{destination}, \text{source}, \text{node}, \text{timestamp}) \).

2.1 Architectural Considerations for MIND

There are three possible ways in which one can structure MIND. A query flooding architecture keeps the flow records at or near monitors and floods each query to every monitor. A

\(^1\) Even though we describe our system in terms of its querying functionality, triggers can just as easily be supported in our system, with minor mechanistic modifications. In what follows, we ignore this distinction.
centralized architecture moves the data to one node (or a cluster for redundancy), and queries are sent to that cluster. Finally, in a distributed architecture traffic monitors store summaries in a specialized routing structure that can forward queries where the summary resides.

The tradeoffs between these architectures are well-understood, but in the network monitoring context we find the distributed architecture to be attractive for the following reasons. Query flooding scales well in that it does not involve movement of flow records, but can lead to poor performance for high query loads since all queries must be evaluated at each node. Furthermore, to make this architecture robust to failure, it may be necessary to replicate flow records in a topologically different location, requiring movement of data. The centralized approach lacks the physical redundancy necessary in an operational network monitoring system. It is always possible to replicate the central flow record repository, but such a system will not be that much easier to manage or provision when compared to a distributed architecture.

A second architectural decision is whether the querying system should be organized hierarchically or in a peer-to-peer overlay. Both these structures have good scaling properties and are natural candidates for the querying system. Hierarchical systems like InSnet [11] organize the namespace hierarchically, and can efficiently support hierarchical wildcarded queries. Such structures can only efficiently support multi-dimensional range queries that are well-aligned with the naming hierarchy. However, traffic data cannot be organized in a manner required by systems like [11]. For example, even if the network prefix is contained in another, packets belonging to the two prefixes may be observed in two completely different parts of the network. For this reason, we only consider peer-to-peer overlays as the basis for our design.

2.2 Design Considerations

While we have argued that MIND should be architected as a peer-to-peer overlay, there are several detailed design decisions that we make in MIND. Most of these decisions, described below, are motivated by the requirements of network monitoring (Section 1).

Routing structure. The design of overlays for supporting range queries has started to receive some attention in the literature [22, 9, 26, 1]. Two classes of approaches have been studied. The first attempts to build support for such queries on top of DHTs. The second stores data on the overlay in a manner that preserves locality—data records are routed to nodes such that records stored at a node are “near” each other in the attribute space. This locality-preserving routing is more naturally suited than the first approach to efficient multi-dimensional querying, since queries can be routed directly to nodes that contain the relevant parts of the attribute space. For this reason, MIND employs the latter approach. In Section 6 we more carefully contrast previously proposed systems with MIND and explain why prior work is unsuitable for network monitoring and anomaly detection.

Pre-filtering traffic summaries. Traffic data can be indexed at various granularities. At one extreme, one can envision building a single index of every TCP flow observed on each link in the network (we consider it obviously infeasible, given the required traffic volumes, to index individual packet headers). While bandwidth may be cheap, per flow data is voluminous (e.g., more than 1 TB/day for the European Research Network) and most of it is likely to be “normal” traffic and therefore uninteresting. For this reason, the MIND system is designed to allow users to dynamically create multiple indices on arbitrary subsets of the dimensions.

This will allow network operators to create one or more indices, and insert into each index aggregated and filtered flows. Such a capability is analogous to the creation of indices in classical databases, and will enable a more bandwidth efficient and responsive system. For example, to detect alpha flows, we can aggregate traffic flows based on (SrcIP, dstIP) and then filter out small flows with low aggregation traffic (in bytes). We have empirically determined that such flow aggregation can reduce the number of flow records significantly. Figure 1 plots the number of flow records after aggregating and filtering the one day's (Sept. 1, 2004) worth of sampled Netflow data from a router on the Abilene backbone. When aggregating the data over a 30-second time window and using a threshold of 50 KB, we obtain almost two orders of magnitude reduction in the number of flow records.

Load balancing. Our choice of locality-preserving hashing can, with a naive design, lead to poor system performance. This is because network traffic data is, in general, significantly skewed. Figure 2 plots the number of flow records that would fit into a 64-bin multi-dimensional histogram constructed on three different indices over a one day traffic summary from Abilene and GÉANT (the indices are described in detail in Section 4). This figure illustrates that without a careful systems design, the amount of data stored at different nodes

---

2. Routers in Abilene's network perform packet sampling with a rate of 1/100, so the actual size of a flow reported by NetFlow might be much larger than the record indicates. Thus, a threshold of 50 KB is conservative enough to capture most alpha flows.
within the network can vary by an order of magnitude.

MIND’s load balancing scheme leverages the fact that network traffic is approximately stationary over diurnal timescales. It uses an approximate histogram of the data distribution from a 24-hour period to govern how data storage is distributed across overlay nodes (this is described in detail in Section 3).

To validate that the distribution of traffic summaries is indeed stationary over different timescales, we analyzed Netfisw records from all routers on Abilene and G EANT over a two-week period (December 1, 2004 - December 14, 2004). We indexed the Netfisw records aggregated (but not filtered, see above) over a 30-second interval on six attributes consisting of the source and destination addresses, timestamp, the total number of bytes transferred, the number of distinct connections, and the average size of a connection). Figure 3 depicts how the distribution of this index varies day-to-day. We define a mismatch metric (Appendix A) to compare multi-dimensional histograms representing the data distribution. Intuitively, this metric estimates the amount of data that needs to be rearranged between nodes to keep the same distribution across days, and represents an upper bound on the rebalancing cost of using the previous day’s distribution to estimate the current day’s distribution. We find that, even with the finest grain histogram, no more than 20% of the data items need be moved from one day’s histogram to another. By contrast, the traffic distribution can vary significantly hour to hour (mismatch close to 1 when the histogram has a 64 or higher granularity (Appendix A)) indicating that the continuous re-balancing proposed in some other systems [3] might be very inefficient. We have also verified that this behavior holds qualitatively for filtered indices built on fewer attributes.

3 MIND Design

Having outlined the design decisions underlying MIND, we now describe the MIND system in some detail. All the functionalities described in this section have been implemented in our MIND prototype, except as noted in the paragraphs below. The next section evaluates our implementation of MIND on PlanetLab using real network flow records.

3.1 Overview

MIND is a distributed system that provides applications with the abstraction of multi-dimensional indices. Applications (or users) can create indices with specific schemas from any MIND node. Any MIND node can insert data into an index, or query an index. Finally, users can delete MIND indices. Thus, in the context of our driving application, a network operator may install a distributed MIND system, then create indices for detecting anomalies. Individual network monitors would preprocess packet-level traces into flow records of the appropriate granularity (see Section 2), and insert them into one or more indices. A network operator (or perhaps individual customers) could script periodic queries that polled for the likely existence of anomalies in different traffic volumes. That script could also be automated to drill down (by issuing a sequence of queries) once a potential anomaly was detected.

Note that although we propose to use MIND for network anomaly detection, the MIND system represents a generic distributed substrate for multi-dimensional range queries. As such, it can be used for other tasks, such as root cause analysis of Internet routing problems, performance analysis or troubleshooting.

The design of the MIND system borrows heavily from the existing literature on overlay networks. MIND organizes a collection of networked nodes into a hypercube overlay. Nodes can dynamically join and leave the overlay in a manner largely transparent to the users. For each index, MIND defines a mapping between the multi-dimensional data-space and the underlying hypercube. MIND’s novelty lies in its de-coupling the data-space mapping from the underlying routing structure. This allows the MIND system to naturally balance the storage load across all the overlay nodes while maintaining the conceptual simplicity of the overall routing scheme.

When an item (e.g., a flow record) is inserted into MIND, its contents are used to hash the record to a position on the hypercube. Traditional hypercube routing is used to store the record at the corresponding node. MIND also replicates the record on a natural “sibling”, in a manner to be described later, to al-
low robust retrieval of data. A query directed to a hypercube in the data space can be hashed (using a method similar to insertions) to a set of positions on the hypercube. This query is then routed to the corresponding nodes which respond directly to the query originator. Conceptually, MIND preserves the locality in the data-space, by storing “related” records on the same node, allowing efficient querying. Thus, the MIND design is predicated on supporting a high-rate of queries, we believe this to be an important design point for network monitoring applications.

3.2 The MIND Interface

The MIND interface provides elementary primitives for applications or users to manage and query their own indexes. `create_index` builds a distributed index in MIND. This primitive takes an XML-based index schema description which includes a globally unique tag for the index. MIND instantiates the named index on all of its nodes. A corresponding `drop_index` is used to remove an index from the system. `insert_record` inserts a multi-dimensional data item into a named index. Finally, `query_index` can be issued to query a specified index.

The MIND interface can be invoked from any MIND node within the network. The interface can be provided using one of many mechanisms, either a library co-resident with the application or using a remote procedure call. This generality in access is needed in the network monitoring application to enable traffic monitors to be able to insert tuples from a nearby MIND node. More generally, the ability to access the MIND interface from any node is crucial to all distributed applications.

Finally, index creation and deletion in a production MIND system must, of course, be appropriately authenticated, as must the insertion and the queries. While our current prototype does not contain support for this functionality, we believe to achieve this.

3.3 Overlay Construction

The logical structure of an $N$-node MIND is a $N$-node hypercube where each vertex corresponds to one MIND node. The choice of a hypercube is not fundamental in MIND; we have not investigated other overlay routing geometries, such as the ring, but we expect that they can be used for MIND, perhaps with different tradeoffs.

Each MIND node uses its hypercube vertex code as its “address” on the MIND overlay. MIND attempts to maintain a balanced hypercube, to the extent possible. A balanced hypercube minimizes the maximum code length of any node, resulting in about $\log N$ neighbors per node. Since the neighbor list is the only state each MIND node needs to maintain in order to make routing decisions, a balanced hypercube will even out the routing table size at all nodes.

![Figure 4](image) The deadlock free concurrent join procedure in MIND.

There are various ways to establish a balanced hypercube with a given set of nodes. One approach, for example, would use a set of well-known hosts to record the current hypercube topology. Other nodes can join the hypercube by querying a well-known host to find an appropriate neighborhood of nodes to join. Adler et al. [1] propose a distributed randomized node join procedure as follows. A host that wants to join randomly chooses a node in the hypercube. In the neighborhood of the selected node including the node itself, it chooses the node with the shortest code to join and becomes that node's sibling node. This join procedure guarantees that the resulting hypercube, after a sequence of node joins, is balanced with high probability.

Because it is amenable to a more robust implementation, we adopted Adler's algorithm in MIND. However, our implementation modifies the original join algorithm to serialize concurrent node joins to the same neighborhood in a manner that does not introduce deadlocks. In our modified case, a node may optimistically accept multiple join requests. However, a new join request preempts an already on-going join process that has not been committed if it joins a shallower node. Figure 4 illustrates the concurrent join process. Consider the case where node $X$ and $Y$ simultaneously join the overlay. In step 1, node $X$ and $Y$ send join requests to node 00 and node 1, respectively, at the same time. In step 2, $X$ joins node 1 with code 00 and $Y$ joins node 00 with code 1. In step 3, unless $X$'s requests have been committed by all nodes $X$ has contacted, only $Y$'s join requests are accepted while $X$ are rejected by both node 00 and node 1 because $Y$ is joining a shallower neighbor. The procedure shown can be easily extended to the cases of more than two concurrent node joins.

3.4 Index Creation and Data Space Embedding

MIND allows the construction of several indices, and maps each index independently onto the overlay. The procedure for creating an index in MIND is relatively straightforward. When an application calls `create_index`, the index creation request, together with the schema for the index, is forwarded on the overlay. When nodes join the overlay, they obtain the current set of defined indices from the neighbor to which they attach. Similarly, when an index is dropped, a message forwarded throughout
the overlay causes all state at each overlay node pertaining to the index to be deleted.

In MIND, the data space associated with an index is stored on the overlay in the following manner. Consider an index built on \( k \)-dimensions. For example, for our alpha flow monitoring application, \( k \) can be 4 (source IP address, destination IP address, flow size, and time). Conceptually, the data inserted into such an index can be described as points in a \( k \)-dimensional space. MIND essentially maps hyper-rectangles in this space into individual overlay nodes, in a manner described below. A key property of this algorithm is that \( k \) can be independent of the dimensionality of the hypercube.

Take the \( k \)-dimensional data space and "cut" it (for now, assume that these cuts divide the data-space equally, but we shall remove this assumption shortly) along each of these \( k \) dimensions using a \( k - 1 \) dimensional hyper-plane. This results in \( 2^k \) hyper-rectangles. At each cut, define one half of the data-space to have a code bit of 0, and the other a code bit of 1. Thus, following this sequence of cuts, each \( 2^k \) hyper rectangle is associated with a bit-string "code". Now, repeat this procedure until the number of hyper-rectangles equals the number of nodes (Figure 5, top left). Then, the data points in a MIND index falling into a hyper-rectangle \( \Lambda \) are stored (we describe later how index data is routed) in the overlay node whose code maximally matches the code of \( \Lambda \). When a node joins an existing overlay and takes over a part of the data-space from a sibling, data already stored in existing indices are not moved from the sibling to the joiner. Rather, the joiner maintains a pointer to the sibling and forwards queries to it. The pointer will be dropped once the data have aged.

When the data distribution is skewed, the number of data items in different hyper-rectangles can be different. In that case, the storage across nodes is not balanced. However, observe that if the global data distribution were known, approximately, then we could "cut" the data space to allow for a more equitable distribution of data items (Figure 5, bottom right, illustrates this). We describe this in greater detail in Section 3.7 below.

### 3.5 Data Insertion

As we have said, users or applications can insert data into MIND from any overlay node. When an application calls `insert_data` at a node, the corresponding MIND instance computes the hyper-rectangle that the data item would fall into based on the algorithm described above. To do this, each MIND node would have to know the hyper-planes that cut the index data space; we describe how a MIND node knows this in Section 3.7. Furthermore, because the cuts will, in general, generate more hyper-rectangles than there are nodes (a node may not know the exact number of nodes in the overlay, but computes an estimate of this from its own code length), the computed code for the data item may not exactly match the code associated with a node. In this case, the data item is stored at the node whose code maximally matches the computed code.

To find this node, MIND uses standard hypercube routing, which greedily routes on the overlay using the computed code. In this procedure, at each step the data item is forwarded to that neighbor whose code maximally matches the computed code until the data item reaches a MIND node which is assigned the hyper-rectangle containing the data item. During routing transients, the routing procedure might hit a dead-end. In such cases, MIND employs a recovery procedure that attempts to find an alternate route for the data item (Section 3.8).

### 3.6 Query Processing

A query on a MIND index is described by a range of values for each attribute of the index (of course, any of the attributes can be wild-carded). Put differently, a query in MIND represents a hyper-rectangle in the data-space. Depending on the size of the query, this hyper-rectangle may be contained within a hyper-rectangle resulting from the data-space cuts described above, or may contain many such hyper-rectangles.

When an application calls `query_index` at a MIND node, that node computes a code for the query in a manner similar to computing codes for data items. However, given that a large query may cover more than one data-space hyper-rectangle, the computed code can be a prefix of the codes assigned to overlay nodes. The query is routed using exactly the same strategy described above for routing data items, but with one important difference. When the query reaches the first node whose associated hyper-rectangle (perhaps partially) abuts the query hyper-rectangle, that node splits the query into sub-queries covering data-space hyper-rectangles. These sub-queries are independently routed using the strategy described below.

For simplicity, results from all sub-queries are directly transferred to the originator rather than being routed on the overlay. The originator can then determine, by examining which nodes responded, when the query response is complete. If a node is assigned a hyper-rectangle in the data-space covered by the query, but has no matching data, it responds negatively to the query.
3.7 Balancing Data Storage in MIND

MIND uses locality-preserving hashing. If the data-space is embedded on the overlay by recursively cutting the data-space evenly across each dimension, then a skewed data distribution (where the data items are unevenly distributed across the hyper-rectangles) can result in significant storage imbalance across the nodes. This imbalance can be a significant practical problem for applications like network monitoring that can store large amounts of data items in different indices.

Classical approaches to re-balancing centralized index data structures usually attempt to do so on the fly, i.e., as data items are inserted into the index. In an overlay based system like MIND, this approach can introduce significant complexity. MIND would have to dynamically re-partition the data space among the overlay nodes, and migrate data items from one node to the other in order to do this. Care must be taken to process queries correctly and to robustly handle node failures during re-balancing. Furthermore, for network monitoring applications, the re-balancing operations must be invoked relatively infrequently to avoid moving large volumes of data between overlay nodes.

For all these reasons, MIND uses a much simpler approach to load balancing that leverages two properties of the data: (i) network traffic data is approximately stationary on diurnal time-scales (see Section 2); (ii) the cuts on the MIND data-space can be scaled so that the hyper-rectangles contain approximately the same amount of data. We now describe the mechanics of this approach.

In MIND, a designated node collects, once a day, an approximate multi-dimensional histogram of the data distribution on each index, by aggregating the individual data distributions observed at each node. These per-index distributions are then sent to each node, which independently computes a balanced cut on the data-space. In such a balanced cut, a hyper-plane cuts a hyper-rectangle such that the number of data items on each division of the hyper-rectangle is approximately the same. The histogram of the data distribution is used to effect this, and the efficiency of load balancing depends upon the granularity of the bins in the histogram.

Having computed the balanced cut, MIND does not attempt to migrate historical data between overlay nodes. Rather, it maintains daily versions of each index (we have deferred an examination of version storage management to future work) separately, and the histogram describing the data distribution on one day is used to store data for the next. This is conceptually easy to do, since a histogram completely defines the balanced cuts and the relevant index versions that should be used will be evident from the query itself (e.g., by examining the time interval described in the query).

Our current prototype does not implement the on-line histogram collection algorithm mentioned above. In the experiments in this paper, we compute the balanced cuts off-line and install them at nodes. However, implementing the on-line histogram collection should not be a major endeavor.

3.8 Robustness to Node and Link Failure

MIND incorporates two classes of mechanisms designed to make the system robust to un-anticipated failures of node or overlay links. The first class ensures query completion in the face of transients on the overlay; the second ensures data availability in the face of node failures.

Dealing with routing transients In our experimentation, we have often observed transient overlay link failures, presumably caused by transient routing failures in the underlying network. Restoring a link by repeatedly attempting to reconnect to the peer node suffices in most cases and has the attractive property that it minimally perturbs the overlay. When successive reconnection attempts fail, the MIND node first attempts to discover whether the peer is alive or not, by routing a probe message on the hypercube overlay to its peer. When a neighbor of the peer receives this message and can attest to the peer’s liveness, the node continues to attempt re-connection. The system attempts to repair the overlay (as described below) only if the peer is found to have failed.

During these transient link failures, a query may encounter a failure in greedy routing. The failed link may have been the only exit from the node towards the destination of the query. There is a fairly substantial literature on the subject of robust hypercube routing (see [11] for examples), but our current implementation uses a rather simple approach. When a query reaches a MIND node at which greedy routing fails, that node sends out an expanding-ring scoped broadcast along the overlay to find another overlay node whose address overlaps the query’s code to an equal or greater extent than that of the broadcast originator. Query forwarding resumes from that overlay node. We have seen several instances of this procedure being invoked during our experiments.

Dealing with node failure The MIND system also attempts to re-configure the overlay after a node failure. Recall, that MIND nodes have addresses on the hypercube. MIND’s recovery procedure is best understood by thinking of the overlay node addresses as forming a virtual binary tree whose leaves are the nodes themselves. On this tree, for example, nodes with addresses 000000 and 000001 are siblings. Furthermore, a node with an address 000010 is said to be on a sibling subtree of these nodes.

In MIND, when a node fails, its sibling takes over the hyper-rectangle associated with the failed node. It does this simply by shortening its code. Thus, in our example, if node 000000 fails, its sibling shortens its code to 00000. If both a node and its sibling fail, then a node in the sibling sub-tree (in our example, the node 000010) takes over. This procedure can be applied recursively. Notice that this recovery strategy may not preserve the balance in the hypercube. We have postponed the design of a balance-preserving recovery strategy pending a better understanding of the frequency of node failure—our sense is that because MIND is intended to be a managed infrastructure (as opposed to a truly peer-to-peer system in which nodes may join and leave at will), our current approach will
suffice.

In MIND, this recovery procedure is closely aligned with our data replication strategy. Notice that the set of nodes that could potentially take over a node's hyper-rectangle are its neighbors on the hypercube overlay. A MIND index's availability can be tuned by carefully replicating data items at these neighbors. With the right choice of neighbors at which to replicate, the failover to replicas is made transparent. For example, let \( m \) be the desired level of replication and \( k \) be the code length of the original node. Then the replication nodes are those neighbors that have common prefixes of length \( k - 1, k - 2, \ldots, k - m \) with the original node, and the system is robust to failure of \( m \) nodes. For example, consider node with code 000000 and \( m = 3 \), then the replication nodes are those with codes 000001, 000010, and 000100.

### 3.9 Software Structure and Implementation Details

We have implemented a MIND prototype using Java. Figure 6 describes the software structure of our prototype. Conceptually, our prototype consists of two separable components, one dealing with network communication (the left half of the figure), and one with data storage (the right half). Our implementation is largely event-driven, but uses three long-lived threads: one for TCP communication, one for the message dispatch, and one for managing storage.

At the lowest level of the communication component is a dispatcher that sends and receive messages. Network maintenance messages including node join and leave, and neighbor status update are processed within the neighbor management and the code management components. Should a node's code (address) change, the code management module sends a request to the data space management module for remapping the multi-dimensional data space to the code. Application data and control messages are delivered to the index management component where messages are further classified and handed off to individual indices. Routing decisions for data and queries are made within the data insertion and query processing modules. The code management module interacts with the neighbor management module to maintain the overlay. The data space management module is also responsible for collecting and distributing histograms and rebuilding indices for lead balancing purposes (in our current prototype, this feature is not yet implemented).

The system also implements several modules that maintain and manage access to the data. Central to this functionality is the database access control (DAC) module, one for each index, which buffers database access requests in a queue and communicates with the local database via JDBC. MIND uses a MySQL backend to store the data items. When invoked, the DAC processes pending insertions received from the network. DAC processing is tuned to handle the relatively high data insertion rates seen in network monitoring applications. Another important function of the DAC is to resolve queries. When invoked, the DAC builds an SQL statement for each pending query and sends it via JDBC to the local database. Upon receiving a response from the local database, the DAC builds one or more response messages and sends them directly to the query originator. Finally, since our robustness mechanisms affect many components, handling failure recovery has not been modularized in our implementation.

### 4 Performance Evaluation

We have experimented with our prototype on PlanetLab [ ]. In general, our prototype is robust enough to handle insertions of tens of millions of flow records over several days. We have conducted several controlled experiments using our implementation, most of them on PlanetLab, to evaluate MIND's behavior at the scale of modern academic backbones, as well as its behavior at larger scales and under greater churn. This section reports the results of these experiments.

#### 4.1 Methodology

For most of our experiments, we deployed a number of MIND instances on PlanetLab nodes (the node deployment strategy is discussed when we describe the experiments). In all experiments, we inserted aggregated flow records from two large backbone networks: Abilene [ ] and GEANT [ ]. The flow records were obtained by processing NetFlow [ ] or eBWD [ ] information collected continuously from all the routers of the two research networks. The details of which datasets we used vary with the experiment, so we describe these below.

On the resulting MIND overlay, we created three indices, each corresponding to a network monitoring or anomaly detection task.

**Index 1** This 3-dimensional index is built from the first three attributes on an aggregated flow record of the form:

\[(dest\_prefix, timestamp, fanout, source\_prefix, node)\]

where fanout determines the number of short connection attempts made by nodes the source prefix to nodes in the destination prefix, and the other attributes are self-explanatory. Such an index is useful for detecting port scanning activity, using a query of the form:
Find all the sources that attempted to connect to more than $F$ hosts in destination prefix $x(s)D$ within the time period $T$.

**Index-2** This 3-dimensional index is also built from the first three attributes of an aggregated fbw record of the form:

$$\langle \text{dest}\_\text{prefix}, \text{timestamp}, \text{octets}, \text{source}\_\text{prefix}, \text{node} \rangle,$$

where $\text{octets}$ denotes the total size of the fbw in bytes. Such an index can be used to find unusually large fbws between aggregates, using queries of the form:

Find all fbws destined for $D$ that have carried at least $O$ octets (or in between $O_1$ and $O_2$) within time period $T$.

**Index-3** Finally, this 3-dimensional index is built from the first three attributes of an aggregated fbw record of the form:

$$\langle \text{dst}\_\text{prefix}, \text{timestamp}, \text{flow.size}, \text{src}\_\text{prefix}, \text{dst}\_\text{port}, \text{node} \rangle,$$

where the $\text{flow.size}$ in a time window is the average traffic send on each distinct connection from a source in $\text{source}\_\text{prefix}$ to a destination in $\text{dest}\_\text{prefix}$ within the time window. Such an index can be used to identify network applications that are using well-known ports to work around fbw rewrites and packet filters (e.g., peer-to-peer applications [14]) or that tunnel their traffic using other application layer protocols to avoid connection charges (e.g., DNS tunneling [3]). The hallmark of such applications is an unexpected amount of traffic to ports on which one would not expect large traffic volumes. Such a pattern can be detected using a query of the form:

Find all the fbws either from source $X$, or to destination $D$, or both, that have fbw size more than $X$ and/or with destination port $P$ within time period $T$.

To insert data into these indices, we create the above-mentioned aggregate fbw records (Section 2). In addition, we also filter out small and "uninteresting" fbw records. Specifically, in our experiments the aggregation time window is chosen as 30 seconds for all three indices. Furthermore, we do not insert fbw records with fanout less than 16 into Index-1, fbw size less than 80 KB into Index-2, and less than 1.5 KB into Index-3. In each case, we deem these thresholds as low enough to not be interesting from an anomaly detection perspective.

Finally, in each experiment, we issued queries that were uniformly sized with respect to all attributes other than the timestamp. For the timestamp, we always used a time interval of the last five minutes. The uniform choice of attributes in other dimensions stresses the system by including some large and some small queries. Having a time window of five minutes is representative of periodic monitoring queries that are continuously assessing the state of the network to determine potential anomalies.

In the following subsections, we measure several aspects of MIND performance: *insertion path length* (the number of overlay hops for tuple insertions), data *insertion latency* ², *query cost* (the number of nodes visited in order to resolve queries), the *query latency*, and the *data and traffic distribution* across nodes and links on the overlay.

### 4.2 The Baseline Experiment

Our baseline experiment was designed to assess the realistic performance of the MIND prototype at the scale of network backbones. To achieve this, we carefully selected 34 PlanetLab nodes, 11 from North America and 23 from Europe. The nodes were chosen such that their geographic locations are close (to within a city granularity in most cases) to the routers of Abilene and GÉANT backbones respectively (recall that our traffic trace data sets also come from these backbones). This way we could emulate the propagation delays that an actual MIND deployment on those networks would experience, since most PlanetLab nodes are on these research networks.

We inserted data collected during a period of three days (September 1-3, 2004) for a total of nearly 9 million fbw records per day. We inserted the aggregated and filtered fbw records at the same timescales as they would have been inserted into the real network: a few filtered fbw records from each MIND node every 30 seconds. Due to the different packet sampling rates configured in the two networks (1/100 in Abilene and 1/1000 in GÉANT), more fbw record tuples were injected from Abilene nodes than from GÉANT nodes.

We ran MIND continuously for three days, but, for brevity, present results for two different hour-long periods (11am to noon, and 11pm to midnight GMT) on each day.

Figure 7 shows the insertion latency for these six time intervals. Most of the insertions were accomplished within a few seconds with the median varying between 1 and 2 seconds and the mean varying between 1 and 5 seconds.

However, the insertion latency distribution has a long tail in many cases, as evidenced by the rather high 99th-percentile numbers. Two factors contributed to the rather long insertion latencies: queuing of data at overlay nodes caused by a transient hotspot, or by network dynamics (failed links or changes in the overlay topology). In particular, we observed that there were several tuples whose insertion took up to several tens of seconds. In one particularly pathological case, queuing delays at successive links delayed the tuple insertion by 48 seconds! We examined the slowest link on this path, plotting the transmission delays (Figure 8) observed on this link over the

---

²For each index, we also need to define an upper bound on the attribute value in order to construct the multi-dimensional histogram. We chose 5024, 2MB and 128 KB respectively for these attributes after analysis revealed that these bounds were exceeded by less than 0.1% of the tuples. For these tuples, we assigned them the largest possible range.

---

⁴To measure this, we used the fact that PlanetLab nodes are synchronized using NTP. As such, our measurements might be affected by NTP errors, typically around tens of ms.
The query cost measures the number of overlay nodes visited by a query, regardless of whether the query is forwarded or resolved by those nodes. Figure 9 shows the overall query cost distribution for all three indices measured during the 11:00 to noon hour on the first day. Recall that except for time interval xed at 5 minutes, the ranges for all the other attributes were randomly selected, so that some queries were for relatively large parts of the data-space. Even so, MIND indexing is quite efficient at preserving the data locality, and over 90% of the queries involved 4 overlay nodes or less while retrieving the results.

Figure 10 depicts the query latency statistics for this experiment. Notice the low median query latency in the figure, about 500 ms. This is very encouraging from the perspective of anomaly detection: it indicates that it might be feasible to deploy a distributed system based on MIND for on-line detection of Internet anomalies.

However, that same figure also indicates that the query latency distribution is skewed—notice the high 90th-percentiles and means. In MIND there are two components to query latency: the time to route the query on the overlay, and the time to send the response directly back to the originator. The former component is qualitatively similar to insertion latencies (a fact borne out by the rough similarity between this plot and Figure 7). Furthermore, in general, the latter component is usually negligible (dominated by the round-trip time between the originator and the responder).

However, there were several instances in our experiments where the query responder could not connect to the query originator as a result of routing outages. In these cases, this component was significantly large. Figure 11 shows the time spent in resolving all queries during the time period from 23:00 to midnight on day 3 at a MIND node. There are two spikes back to back for two different indices corresponding to attempts by the query responder to contact the query originator. During this network outage, it took 45 seconds to re-establish the overlay links. This long delay is an artifact of our implementation, which attempts to repeatedly re-establish an overlay link several times before deciding to find an alternate route. We can improve this significantly. Additionally, one of these queries was queued behind the other and suffered an additional delay, also an artifact of the MIND implementation where query database access is not interleaved with network transmission of query results.

We plotted the insertion traffic over each link over one day (Sep 1, 2004), as shown in Figure 12. This distribution is not perfectly balanced because of the imbalance in traffic volume from Abilene and GÉANT nodes. Regardless, the traffic on any link is far less than what a centralized solution would have
Figure 11: Query processing delay per query at the query hotspot node.

Figure 12: The number of tuples traversed on each link, Sep 01, 2004.

Figure 13: The distribution of data storage across MIND nodes.

Figure 14: The cumulative distribution of insertion latency on the 102-node MIND overlay.

4.3 A Large-Scale Experiment

To explore the performance of MIND at a larger scale, we deployed the MIND prototype on 102 nodes. Unlike our baseline experiment, these nodes were arbitrarily chosen but were distributed across North America and Europe. During the course of our experiment, several nodes failed and rejoined the overlay so that the actual number of operational nodes varied from 70 to 102. In addition to scaling the network size, we also scaled up the network traffic by inserting aggregated 1MB records from three days' worth of Abilene and GEANT traces (about 11 million records of Index-1) at the rate of 1 record per second per node.

The results we obtained from this experiment are qualitatively similar to those obtained in the baseline experiment. Nearly 90% of insertions incur less than 5 hops on the overlay, but some of the insertions incur 12 hops more than the network diameter because MIND re-routed those insertions around failures. Furthermore, 90% of the nodes visited less than 5 nodes, and at most 12 nodes were visited by any query. We have omitted graphs describing these results for reasons of space.

We focus on the distribution of insertion latencies (the query latency distribution is qualitatively similar). Figure 14 describes the insertion latency distribution across the entire experiment. Notice that while the median latency is below 1 second, the tail of the distribution is long. As in our baseline ex-
4.4 Robustness

Finally, we evaluated the robustness of the MIND design to node failure. To do this, we deployed a 102-node MIND prototype on a local cluster of nodes, with several instances of MIND running on each physical node. This deployment enabled us to fail individual nodes in a controlled fashion and observe the availability of data in MIND, at various levels of replication (Section 3.8).

Into this network, we inserted all three days' aggregated and filtered flw records for Index-1, but varied the degree of replication and measured the fraction of successfully completed queries at varying levels of random node failures. Figure 16 shows the results of our experiments when using 0, 1, and "full" replication per flw record. In full replication, each item is replicated at all overlay neighbors.

Without replication, fraction of successful queries decreases almost linearly with the number of node failures. With one replica per data item, MIND can survive 15% of node failures without loss in data availability. With full replication, MIND can easily survive over 50% node failures without loss of data availability. Of course, replication storage and transmission cost scales linearly with the degree of replication.

5 MIND and Anomaly Detection

MIND is intended to support flexible examination of traffic volumes in order to detect potential network anomalies. The results of MIND queries can be used to drill-down into the data, i.e., trigger a more detailed examination of flw records or packet traces. To get an initial sense of how well MIND achieves this goal, we used results from Lakhina et al.'s work on the efficacy of off-line centralized anomaly detection [14]. They detected several anomalies on the Abilene backbone on December 18th, 2003. These anomalies were of three types: alpha flws, DoS attacks and port scans. A detailed discussion of [14] is beyond the scope of this work.

We conducted an experiment to see if (and how well) MIND is able to capture the same anomalies. We constructed an 11-node MIND overlay on PlanetLab congruent to the Abilene backbone topology, and built two indices on this overlay (Index-1 and Index-2 from Section 4.1). We then inserted aggregated flw records from about 25 minutes worth of Abilene backbone traces during which several anomalies were observed by [14]. Finally, we issued queries on traffic volumes (multi-dimensional hyper-rectangles) circumscribing the observed anomalies. Our goal was to see how quickly MIND responded to those queries (the average response time when the query is issued from every node), and how closely our traffic volumes circumscribed the records corresponding to the anomaly.

To detect DoS attacks and port scans, we issued a query on Index-1 of the form:

Find all flw records whose fanout is greater than 1500, and whose timestamp fell within a specific 5-minute interval.

To detect alpha flws, we issued a query on Index-2 of the form:

Find all flw records whose total size fanout is greater than 4000000, and whose timestamp fell within a specific 5-minute interval.

Figure 17 summarizes the results of our experiment. The first column identifies the time of occurrence of the anomaly.
<table>
<thead>
<tr>
<th>Anomaly Time</th>
<th>Result size</th>
<th>Actual size</th>
<th>Average Response time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:45</td>
<td>43</td>
<td>2 alpha flows</td>
<td>2.09</td>
</tr>
<tr>
<td>13:30</td>
<td>38</td>
<td>2 alpha flows</td>
<td>1.38</td>
</tr>
<tr>
<td>15:55</td>
<td>55</td>
<td>2 alpha flows</td>
<td>1.47</td>
</tr>
<tr>
<td>19:50</td>
<td>10</td>
<td>2 DoS, 1 scan</td>
<td>0.81</td>
</tr>
<tr>
<td>19:55</td>
<td>13</td>
<td>2 DoS</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Figure 17: Real world anomaly detections using MIND indices

In each case MIND returned a super-set of the flow records that constituted the anomaly. Notice how closely we can constrain the traffic volume. In all cases, the number of flow records returned by the MIND query is relatively small (tens of flow records). Of course, in our experiment, we had the advantage of knowing what we were looking for. In practice, a network operator would arrive at this by programmatically querying progressively smaller traffic volumes. Furthermore, note that the average response times are encouraging, on the order of a second. Finally, a useful by-product of a MIND query response (which illustrates the fact that MIND essentially correlates data from different nodes) is the exact set of network monitors (Abilene routers) which observed the anomalous traffic. For example, for the 2 DoS flows captured in time window 19:55, the returned tuples show that path taken by the DoS flows included the Abilene backbone routers in the following locations:

- DoS 1: CHIN, DNVR, IPLS, KSCY, LOSA, SNVA
- DoS 2: CHIN, IPLS

6 Related Work

Most closely related to MIND is the recent literature on support for range searches using overlays [1, 18, 17], or geographic embeddings [19, 17]. Mercury [19] replicates data on $k$ separate Chord rings where $k$ is the number of indexed attributes. It can resolve multi-dimensional range queries by carefully querying one of the rings based on the selectivity of the data. It dynamically re-balances the storage load on the overlay by using random sampling to approximate the overall data distribution. Mercury's high degree of replication and dynamic load balancing mechanisms may not be suited to the network monitoring application, for reasons discussed previously.

SkipIndex [20] is a distributed implementation of the SkipGraph [18]. Unlike MIND, which decouples the data-space embedding from the overlay construction, in SkipIndex the entire index may need to be rebuilt if the data distribution changes significantly. For high volume insertions as in traffic monitoring, for example, MIND's approach of re-inserting data into a new version of the index might perform better.

PIIT [21] is a qualitatively different approach that attempts to build support for range queries on an underlying DHT overlay. This approach is powerful from a systems perspective, since it can leverage existing infrastructure. PIIT has been shown to support range searches on a single-key, but, to our knowledge, there does not exist literature describing a detailed design and extensive evaluation of PIIT support for multi-dimensional range searches.

MIND is heavily influenced by the DHT literature [17, 30, 29, 31]. Its geometry resembles that of a class of DHT systems that use hypercube structures but differs significantly from DHT systems in the way data items are hashed to nodes.

Finally, prior work on multi-dimensional indexing in wireless sensor networks (DIM [22]) is related to MIND. However, unlike MIND, DIM relies on a geographic embedding of the data space. This difference completely changes the overlay management and data routing mechanisms and, as such, MIND is a qualitatively different from DIM.

Ongoing research on distributed database query engines such as PIER [23] focuses on sophisticated query execution (e.g., joins). Such a system might be components of the network monitoring query sub-system discussed in Section 2. Recent work on anomaly detection is also related to our work [17, 24]. Systems like these can be built on top of MIND, since MIND indices can first be used to detect possible anomalies, and these systems can then be used to improve the robustness of the detection by detailed traffic analysis from a subset of the traffic monitors.

Our work on MIND relies on Adler et al.'s [25] proposed design of a hypercube hash table which achieves optimal load balancing on key space (Section 3). There is also work in the theory literature on on-line load balancing [26, 27]; however, this literature does not apply to multi-dimensional data [28].

7 Conclusions and Future work

We have described the design and preliminary evaluation of the MIND system. Our initial evaluations suggest that it might be feasible to use MIND to perform on-line near real-time anomaly detection in a large network. However, much remains to be done to achieve this goal. Understanding how to design queries for periodic monitoring, how to automate the process of drilling down to find potential anomalies, then implementing the actual algorithms that examine detailed traffic traces based on MIND's query results, are all interesting challenges. Equally interesting are some open issues in the overlay design: fast re-routing to avoid re-connection latencies, more sophisticated but practical alternate path routing on the hypercube, and automated histogram collection and version management.

References


A The Mismatch Metric

There are many ways to compare distributions or histograms. One that is particularly intuitive for our application is the mismatch between two histograms, defined as follows. For a d-dimensional index, we partition it into kd (k is called histogram granularity) equal-sized bins. For bin x (1 ≤ x ≤ kd), we define

\[ I_k^d(i, x) = \text{number of Day-i tuples falling into bin x}. \]

The mismatch between the data distributions of Day-i and Day-j, \( MF_k^d(i, j) \), is then defined as

\[ MF_k^d(i, j) = \frac{\sum_{x=1}^{kd} |I_k^d(i, x) - I_k^d(j, x)|}{2}. \]

When \( kd \) is high enough, a daily balanced data allocation can be generated by directly assigning the bins to network nodes, and \( MF_k^d(i, j) \) is the upper bound of the re-balancing cost to have Day-j data allocated evenly among the nodes with Day-i data allocation scheme as the base. One can analogously define the mismatch at different timescales (e.g., hourly).