An In-memory Database for Prototyping Anomaly Detection Algorithms at Gigabit Speeds

by

Travis Friesen

A Thesis
Submitted to the Faculty of Graduate Studies
of the University of Manitoba
in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg

Copyright ©2013 by Travis Friesen

Abstract

The growing speeds of computer networks are pushing the ability of anomaly detection algorithms and related systems to their limit. This thesis discusses the design of the Object Database, ODB, an analysis framework for evaluating anomaly detection algorithms in real time at gigabit or better speeds. To accomplish this, the document also discusses the construction a new dataset with known anomalies for verification purposes. Lastly, demonstrating the efficacy of the system required the implementation of an existing algorithm on the evaluation system and the demonstration that while the system is suitable for the evaluation of anomaly detection algorithms, this particular anomaly detection algorithm was deemed not appropriate for use at the packet-data level.

Acknowledgements

I could not have completed this thesis without the help of many important individuals.

First, I would like to thank my advisors, Bob McLeod and Paul Card, for their advice, guidance, and immense patience.

I would like to thank my colleague Mike Himbeault, co-designer of ODB, and all-around math wizard.

A big thanks to the folks at MERLIN, especially Jared Bater, for their help and cooperation in collecting the data set.

I would also like to thank my editing team, namely Patricia Sumter, Adam Kroeker and Christine Plett, for their eagle-eyed suggestions and corrections.

A special thanks to TRTech and the staff there, for supporting my graduate studies, and providing the resources necessary to complete my thesis.

Lastly, a big thank-you to all my friends and family. This thesis would not have bene possible without your support and encouragement.

Contents

Acknowledgements **i**

1 Introduction **1**

1.1 Goals . . . 2

1.2 Organization . . . 4

2 Background **5**

2.1 Motivation . . . 5

2.1.1 Security Information and Event Management (SIEM) . . . 6

2.1.2 Advanced Persistent Threats (APT) . . . 7

2.2 Shortcomings of Sampling . . . 8

2.3 Databases for Packet Storage and Indexing . . . 9

2.4 Related Tools . . . 12

2.4.1 Snort . . . 12

2.4.2 TippingPoint . . . 13

2.4.3 Wireshark . . . 13

2.5 Summary . . . 14

3 System Design and Implementation **15**

3.1 Design . . . 16

3.1.1 Datastore design . . . 19

3.1.2 Indexes . . . 22

3.1.3 Performance tuning . . . 24

3.2 ACID Compliance . . . 26

3.3 Limitations . . . 28

3.4 Possible Deployment 29
3.5 Future work 30

4 Data Sources **32**
4.1 Unsuitable Data Sources 32
4.1.1 DARPA Datasets 32
4.1.2 Other Existing Data Sources 34
4.2 Construction of Data Source 35
4.2.1 Injected Attacks 37
4.3 Dataset Shortcomings 40

5 Entropy as Anomaly Detection **42**
5.1 Entropy of Network Traffic 42
5.2 Principal Component Analysis 43
5.3 Algorithm Evaluation 45
5.4 Implementing Algorithm on ODB 47
5.4.1 Advantages of ODB 47
5.4.2 Workflow 47

6 Results **50**
6.1 Evaluation of Candidate Algorithm 50
6.2 Performance of ODB 55
6.2.1 Ease of Use 56
6.2.2 Processing Speed 56
6.2.3 Memory Efficiency 57

7 Conclusion **59**

A Network Attack Schedule **61**
A.1 Host A 61
A.2 Host B 64
A.3 Host C 66

List of Tables

2.1 Comparison of databases: Time to insert X packets (seconds) 10

3.1 Comparison of libraries: Time to insert (seconds) 26

6.1 Timebins with SPE exceeding threshold 53

6.2 Comparison of databases: Time to insert (seconds) 57

6.3 Comparison of in-memory databases: Memory usage (Bytes) 58

A.1 Attacks from Host A 63

A.2 Attacks from Host B 65

A.3 Attacks from Host C 67

List of Figures

3.1 System Diagram 18

4.1 Network Diagram 36

5.1 Principal Component Analysis 44

5.2 Algorithm Flowchart 48

6.0 Principal Component Analysis of Normal plus Anomalous Traffic . . 52

6.1 Squared Prediction Errors (SPE) 54

Chapter 1

Introduction

Mirroring Moore's law of computer performance growth, computer networks have grown larger while network speeds have gotten faster [1, 2]. The result of this is that computer networks push more data and contain more hosts communicating with more destinations than ever before, and will only continue to grow larger and faster in the future. Increases in computer performance as predicted by Moore's law offer no solution, as network speeds and capacity are increasing at a similar rate, or, in some cases, an even faster rate, than computer performance [1, 3]. Managing these networks becomes increasingly more difficult for system administrators, who have had to increasingly rely on automated tools to ensure the stability and reliability of their networks.

As computers and the internet have grown more ubiquitous, so have the number and ability of malicious users, hackers, spammers and creators of worms and viruses [4]. System administrators use a number of tools to protect their networks from these malicious users, such as intrusion detection systems (IDSs), firewalls and monitoring tools, like Wireshark.

But the larger size and faster speeds of modern computer networks have made

manually sifting through network traffic tedious and unfeasible. Furthermore, these advances are beginning to stress the limits of what these monitoring tools can cope with [4]. On top of this, developing and evaluating new anomaly detection algorithms and tools consumes time and resources, and researchers often have to redo work that has already been done.

The race between would-be intruders and security analysts is an ever-evolving game of cat and mouse [4]; in fact, some analysts believe that the hackers and malicious users are ahead of the analysts [5]. As intruders develop new techniques to infiltrate networks, analysts develop new tools to stop or detect them, which in turn causes intruders to develop new, more advanced techniques to avoid these new tools, and the cycle continues ad infinitum.

A new framework and a set of tools are needed which would simultaneously allow for the rapid development and evaluation of intrusion detection algorithms, but while being capable enough to allow these new algorithms to process data at the sorts of speeds that are being seen on modern networks.

1.1 Goals

With the former in mind, the goal of this research was to design a system for evaluating and prototyping network traffic analysis algorithms at gigabit or better speeds on commodity hardware. The phrase 'commodity hardware' refers to the sort of computers typically bought by small to medium sized firms for less than 5 thousand dollars, and restricts the use to a single, if somewhat powerful, host [6]. Though this goal would be easy to achieve on a larger cluster or supercomputer, not all researchers or firms have access to that kind of computing power, or the resources to afford it. It

is anticipated that a successful use of the system to test an algorithm could evolve to deployment of the algorithm with the system; since the work of developing the new algorithm is already done using such a system, researchers would be motivated to deploy the algorithm as-is instead of adapting it to a new system. In this case, being able to use lower cost and commonly available hardware is highly desirable, in order to deliver a product at lower cost. Lastly, by achieving the goal on slower hardware, the system could, in the future, scale up to take advantage of faster and more expensive machines to perform analysis on even larger datasets.

In summary, the design goals for this system are:

1. The ability to quickly adapt new anomaly detection algorithms and methods for anomaly detection to a variety of similar datatypes
2. The ability to quickly store, index and process incoming data at gigabit speeds on commodity hardware
3. The ability to handle the large quantities of data generated by gigabit links

Designing the system necessitates testing it and proving its merit. In order to achieve this, it is necessary to implement a working algorithm on the system and determine its efficacy. This thesis implements an algorithm similar to that used by Lakhina et al. [7, 8, 9, 10], with some adaptations based on missing information and differences in hardware and data. This algorithm is discussed in detail in chapter 5.

Lastly, to evaluate the algorithm, which is the purpose of the designed system, it is necessary to create a dataset with known anomalies on which the algorithm can run. MERLIN, a small Internet Service Provider (ISP) that provides internet to most of Manitoba's schools, assisted with the construction of such a dataset, by capturing the internet traffic going to and from one of its clients, and allowing the injection of

anomalous, or malicious, internet traffic at known times. Evaluation of competing datasets and construction of the dataset used in this research is discussed in chapter 4.

1.2 Organization

This thesis is split into seven chapters.

Chapter 2 provides background and an overview of existing solutions, as well as a demonstration of their inadequacy for the task.

Chapter 3 discusses the design and implementation of the system, as well as the steps taken to improve its performance, the system's limitations, and some potential future work.

Chapter 4 discusses the lack of suitable data sources for testing, and the methodology used to create a new data source with known anomalies.

Chapter 5 discusses the algorithm chosen for testing the system.

Chapter 6 discusses the results obtained from running the algorithm on the system. It also contains results comparing the designed system to other popular database systems, both in speed and memory efficiency.

Chapter 7 concludes the thesis, summarizing and providing a final discussion on the points herein.

Chapter 2

Background

This chapter discusses some of the project's background, and explains the motivation behind the design decisions. It provides a look at existing solutions, why they are inadequate, and how a new system can augment existing solutions.

The result of this research was the Object DataBase, or ODB. ODB was designed for the rapid prototyping of anomaly detection algorithms at gigabit speeds, but has many potential applications in the security research and analysis field. A more complete discussion of the design and features of ODB can be found in chapter 3.

2.1 Motivation

This section discusses some of the motivations for bringing about ODB, specifically how it can interact with and augment existing tools to handle modern threats.

2.1.1 Security Information and Event Management (SIEM)

A medium to large sized network generates a great deal of metadata about its health and behaviour. SIEMs are an attempt to draw in this diverse information and tie it together in a fashion meaningful to the network operators, particularly in relation to log management [4, 11, 12]. SIEM engines use algorithms to attempt to provide correlation of this large corpus of data.

Because the rise of SIEMs as a security management tool is a relatively recent development, SIEMs are lacking in many important capabilities and features. Specifically, [4] notes that SIEMs lack adequate data collection, are overwhelmed attempting to manage data and are unable to make sense of the data collected.

In the words of one security researcher:

> Searching data across TBs (terabytes) of data is the most important problem every organization faces. How do our SIEM solutions solve this? By using some sort of Databasing and Indexing. All the databases today (Read Oracle/SQL/MYSQL/PGSQL) are all limited in terms of handling such randomly formatted, high volume feeds, thereby rendering long term searches, trend analysis etc a slow, frustrating and time consuming job. [12].

In addition to this, SIEMs have no capability to do packet- or flow-level analysis of network traffic. Because of the subtly of many network anomalies and intrusion attempts, many attacks may not register at the log-level, and will not be noticed by SIEMs.

The key theme in security research is that SIEMs lack the ability to quickly and efficiently sort through large datasets. A fast, highly-efficient datastore could help

SIEMs to manage this influx of data, as well as provide for new capabilities.

2.1.2 Advanced Persistent Threats (APT)

A new concept in security research are Advanced Persistent Threads (APTs). This category of threats refers to long-term attempts to compromise a network's security by an organization with significant resources, patience and motivation to mount such an attack. A recent example of an APT realized was the attack on Iran's nuclear program by the Stuxnet worm.

Targetted attacks are on the rise. According to Symantec, attacks of this type increased a staggering 81% from 2010 to 2011 [13].

Because of the limitless timeframe on APTs, the patient attacker can perform very low-intesity attacks over a long period to gain access to the target network, in contrast to the high-intensity driveby attacks that are performed by most criminals looking to make a quick buck. As a result, the attacker can hide their activities in general network traffic. Essentially, they can fly in under the radar, disguising their behaviour over a large set of data, making correlation and discovery of their activities difficult for modern tools.

By increasing the efficiency and of modern data storage systems, analysts can look more closely at larger datasets that go further back in time, increasing the odds of finding an intrusion attempt.

2.2 Shortcomings of Sampling

A common approach to address the problem of increasing network speeds and excesses of data is to use packet or flow sampling. Essentially, in order to reduce the computational complexity of large amounts of data, developers simply sample the incoming packet feed, only addressing one out of every 100 or 1000 or more packets, flows, or other data quanta.

In many cases, developers and scientists have had good success with this approach. In general, algorithms that rely on general network statistics are little affected by sampling. Provided enough packets are sampled, the distribution of sampled packets should closely follow that of the entire network [14].

But sampling is not a universal panacea. No matter how frequent sampling is done, some information is lost in the process. Signature-based algorithms have a particular problem using sampling, because many attacks are stateful (that is, information later in the attack is dependant on information from earlier), and information missing from the attack can lead to a missed opportunity. Brauckhoff et al. suggest that although most statistical measures remain intact, sampling can grossly bias the number of flows [14].

Also, by omitting occasional packets, it is theoretically possible for an attacker to hide his or her malicious traffic by timing the transmission of malicious packets. When studying and implementing computer security, it is important to assume that an intruder has complete knowledge of a network's setup and infrastructure, including location, version and configuration of any intrusion detection tools. This is of particular concern when studying Advanced Persistent Threats (APTs) — threats with sufficient time, resources and motivation to make the effort to discover and exploit

any potential weakness in a secure system.

Although sampling is a useful tool, it is desirable to be able to analyze complete captures whenever possible. Increasing the performance of the analyzing algorithm and the system it is running on allows for analysis of complete captures at faster line speeds.

Increased system performance is still desirable, even if sampling is necessary, because it means that more packets can be sampled, providing a truer picture of the network behaviour.

2.3 Databases for Packet Storage and Indexing

The natural software for storing and correlating large quantities of data is a database, and much research has been done in the field of databases with the goal of making them both fast and dynamic.

However, most database systems are focused primarily on data integrity and long-term storage, concerns which are less crucial when doing live analysis. Traditional databases store their information on hard disks; this incurs a significant performance penalty, which buffering can only partially ameliorate.

As a result, existing database solutions, such as MySQL or PostgreSQL, are either too slow to store and correlate internet packet data at line speeds, or have other limitations that render them unsuitable.

To illustrate this, four popular database solutions were compared in a simple performance test. In this test, a basic database structure was created for typical packet data, storing the traditional IP 4-tuple (source and destination IP and port), as well as the payload, with indexes on each of the metadata items: source and

destination IP address and port. The implementation was done using each library's C API, and utilized common performance tuning options for each database, such as grouping inserts into transactions. Each entry in Table 2.1 is the average time over three runs it took for a particular database to insert that many packets. In all cases, variance was less than 5%.

Database	50k	100k	500k	1M	5M	10M
MySQL	7.73	15.91	82.08	167.27	1055.39	2045.25
PostgreSQL	18.25	34.61	166.10	332.09	2874.71	11364.23
SQLite	1.00	2.08	17.86	119.25	12194.45	39982.88
SQLite (in-memory)	0.75	1.63	8.95	17.96	103.39	217.53

Table 2.1: Comparison of databases: Time to insert X packets (seconds)

The test machine was hosted on a XenServer virtual machine, and allocated 12 of 16 virtual cores (using 8 Xeon 5500s with hyper-threading) and 32GB of RAM.

Experiments on MERLIN, the aforementioned ISP, and TRTech's, a small technology research firm, networks reveal an average packet size of about 600-800 bytes; a fully saturated, full-duplex link will generate 300,000 to 400,000 packets per second. This means that a solution would need to process 10 million packets in about 25-30 seconds. As can be seen in Table 2.1, none of the databases tested come close to meeting this requirement. The best performance, by SQLite using an in-memory database, is still 8-10 times too slow. Further, all solutions demonstrate some degree of sub-linear scaling. That is, they become less efficient as more entries are added. This would hinder their ability to be used on very large datasets, as they would become slower and slower until they became virtually unusable.

MySQL and PostgreSQL both support a method of reading data from a file and inserting it directly into the database, called LOAD INFILE and COPY FROM, respectively. While some people have been able to achieve speeds over 200,000 insertions

per second using these methods [15], this is still too slow for a solution. Furthermore, neither method supports the insertion of binary data, such as would be found in an IP payload, without additional processing that would significantly reduce the database's throughput [16].

An alternative to traditional disk-backed databases are in-memory databases. Several popular solutions exist, including Redis [17] and Tokyo Cabinet [18]. Many of them, such as Redis, are designed with flexibility, not performance, in mind. Of the solutions that are designed with the right focus, performance is still inadequate for gigabit line-speeds. Many of the in-memory databases are not memory efficient, requiring multiples of the data size to properly store an item. This limits the total amount of information that can be stored at one time. For a comparison of the efficiency of in-memory databases, and how they stack up against ODB, please refer to section 6.2.3.

Other candidates include Apache Cassandra, which scales well to clustered deployments, but requires many hosts and much hardware to achieve required speeds. Benchmarks for Cassandra report requiring 48 quad-core nodes to achieve 175,000 writes per second [19] (albeit with a replication factor of 3). Other solutions, such as OpenLDAP, are designed with "write-once" behaviour in mind; that is, they greatly favour read-performance at the expense of write-performance [20, 21]. This is an unacceptable trade-off in light of the requirements previously laid out.

Good speeds are often achieved with key-value stores, such as Tokyo Cabinet, where the simplicity of the design allows for very quick insertion speeds. However, key-value stores provide little in the way of features for indexing or collating data; they are designed simply for fast storage and retrieval, and are therefore unsuitable for any sort of advanced analysis.

In summation, other solutions frequently incorporated features that were not needed for the problem space, such as long-term data persistence, and the additional complexity reduced performance or memory efficiency to unacceptable levels. In other cases, certain key functionality was missing, and could not simply be tacked on in a satisfactory manner. In still others, systems were designed for large deployments or clusters, such as Apache Cassandra or Hadoop, requiring hardware not available to most researchers or firms. For these reasons, the creation of a new system was desired.

2.4 Related Tools

This section details a few of the more popular tools that are similar to the system designed for this thesis, and highlights the differences between them and ODB.

2.4.1 Snort

Snort [22] is an open-source intrusion detection and prevention system. It is extremely popular, has seen wide deployment, and even been hailed by experts as one of the greatest open-source projects of all time [23].

Snort is an event-driven system. As a packet comes in, it is briefly analyzed by Snort's decoders. If the packet is deemed to be relevant, it is handed off to one or more preprocessors. These preprocessors can either perform work preparing the packet for the detection engine, or perform their own analysis on the packet itself. Through this system, Snort can provide both signature-based and statistical analysis of network traffic. If certain conditions are met, Snort can perform a variety of actions, including alerting an administrator, or even blocking malicious traffic.

While a powerful and fast system, Snort does not provide a generic packet storage, indexing and retrieval system, such as ODB, to its preprocessors. This limits the data available to developers, and is not generally well-suited to the rapid development on new anomaly detection algorithms.

It might be possible that ODB can be combined with Snort to provide such an infrastructure, but that analysis is outside the scope of this thesis.

2.4.2 TippingPoint

TippingPoint [24] is an intrusion detection system provided by Hewlett-Packard (HP). TippingPoint provides both signature-based attack detection, as well as management of a reputation database and blacklist, maintained and updated by the Digital Vaccine Labs at HP. While TippingPoint exports a portion of its data to a MySQL database, potential developers are limited in the data made available to them, and how it can be accessed, making TippingPoint poorly-suited to research or development of novel anomaly detection algorithms.

2.4.3 Wireshark

Wireshark [25] is an open-source and cross-platform suite for manually inspecting network packets. While Wireshark supports powerful protocol decoding and many useful graphical user interface (GUI) tools and features, Wireshark is not designed with writing automated analysis algorithms in mind. Wireshark does include support for scripting its protocol detectors, using the programming language Lua. However, the performance of Wireshark is not acceptable for live analysis of network traffic, and the use of Lua only slows it down more. For example, in testing, it took Wireshark

about 15s to do basic processing on a file containing 1.7 million packets, or about 100,000 packets per second: too slow for our purposes.

2.5 Summary

In general, there is a need for a high-speed, efficient way of storing and analyzing network data. Not only would such a system speed research of new algorithms for intrustion and anomaly detection, but many existing tools, such as SIEMs could take advantage of such a system to expand their capabilities. Further, while databse solutions abound, none provide adequate performance, efficiency or capabilities to fill this role.

Chapter 3

System Design and Implementation

Designing a new system for evaluating anomaly detection algorithms begins with identifying the software requirements for such a system. As discussed earlier, these are summarized as:

1. The ability to quickly adapt new algorithms and methods for anomaly detection to a variety of similar datatypes
2. The ability to quickly store, index and process incoming data at gigabit speeds on commodity hardware
3. The ability to handle the large quantities of data generated by gigabit links

The system created was named the Object DataBase, or ODB. While the system stores discrete units of information, like objects, and was written in an object-oriented language, C++, its name should not be taken to mean that it exclusively stores objects created in the C++ or object-oriented style. Any discrete block of contiguous

information or memory — an 'object' in the looser sense — can be stored and accessed by ODB.

ODB does not represent a complete intrusion detection system, but instead is a system that emphasizes the rapid research and development of novel intrusion detection techniques. It is intended to be a useful tool to security analysts and researchers, that while meant for the lab, could one day find its way into production environments.

This chapter discusses the design of ODB, and how it meets and exceeds these requirements.

3.1 Design

The ODB is an in-memory database which can quickly store and sort items by multiple simultaneous criteria. In order to meet the speed requirements, an in-memory database was chosen over a traditional disk-backed store.

A single, modern, consumer-grade harddrive can attain sequential write speeds of over 125MB/s. This means that only 2 or 3 such harddrives, arranged in a RAID 0 configuration, would be sufficient to record packet dumps at gigabit line speeds. But the problem with harddrives as a packet storage solution is not one of throughput; we need to do more than just store the data. The data needs to be sorted in a meaningful matter and be easily and quickly accessed for processing, and modern harddrives are not sufficient for the task. A typical modern harddrive takes about 5ms to perform a random read access. In other words, only 200 packets could be indexed every second, greatly reducing the speed of any analysis that does not access packets in a sequential fashion. And while arranging harddrives in a RAID configuration can dramatically increase their throughput, little can be done about random access times.

Higher-end harddrives, such as solid state drives (SSDs) are able to achieve access speeds as low as 30 micro seconds, two orders of magnitude better than traditional harddrives. However, this still limits processing to 30-40k packets per second. By comparison, the main memory of a typical PC has access times around 50 nanoseconds, three orders of magnitude faster than the best SSDs currently available. This would allow for 20 million packets to be accessed under ideal circumstances, and if other constraints were ignored.

The system consists of two main, interchangeable components: a datastore for storing the data in memory, and a set of index tables for correlating and analyzing the data. While the datastore stores data in an efficient but unordered fashion, the index tables point to individual pieces of data in a way that is meaningful to the user.

All work done on the system is broken into work units, which are stored in a work queue. This allows for very efficient allocation of computation resources across multiple threads.

All data structures were designed with not only computational speed in mind, but also memory efficiency. This is important when designing an in-memory database, because the amount of data that can be stored is limited by system memory; therefore, the memory overhead of the database must be minimized.

To better illustrate how the various components of ODB work together, Figure 3.1 provides a high-level view of the system's components and traces out the steps performed when a new item is added to a running system. These steps are:

1. A new item is submitted via the API. A new work unit is created and added to the work queue.

2. A worker thread wakes up and receives the work unit from the queue.

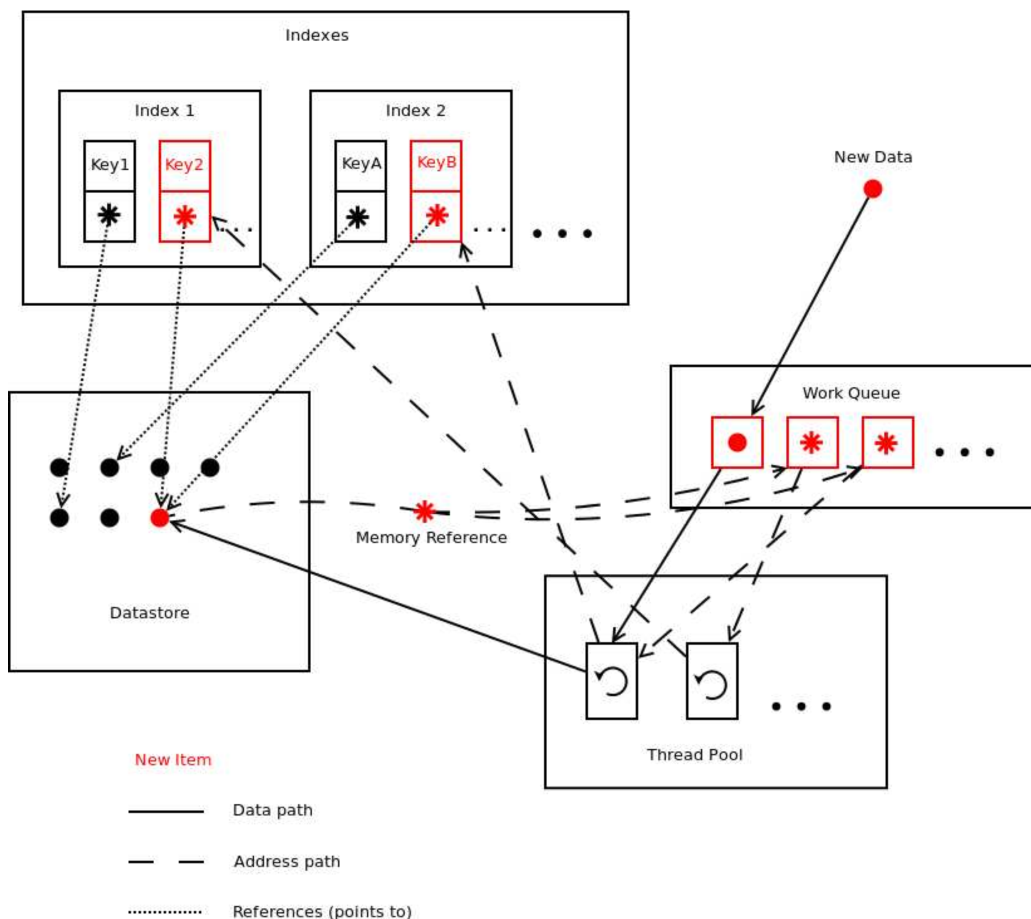


Figure 3.1: System Diagram

3. The worker thread performs the tasks in the work unit. In this case, it adds the item to the datastore.

4. Before finishing up, the worker thread adds any necessary additional tasks to the work queue. Here, it adds a new index update task for each index the item is to be indexed by, in this case 2.

5. Worker threads are activated as necessary to complete the update.

6. Each worker thread updates the appropriate index with the key and reference to the new item.

3.1.1 Datastore design

ODB currently implements two different datastore designs, a bank datastore (BankDS), and a linked list datastore (LLDS). For most situations, the BankDS is the recommended datastore, as it generally outperforms the alternative, though it has several limitations, this necessitates the design of the LLDS for when those limitations are too constraining.

Indirect versions of these datastores are available, for use when the power of the datastores and indexes are wanted, but the element(s) to be indexed already exist in memory. Indirect datastores (IDSs) contain only a pointer to the relevant item.

IDSs also allow for sub-indexing while minimizing the use of memory. A type of query can be made which returns a sub-set of the elements in a datastore; to track these elements, and allow for sub-querying, the elements can be stored in an IDS without having to copy each element individually. In the future, such a system will allow for a user to make dynamic queries on the data, returning sub-sets and sub-sub-sets of data.

Bank Datastore

The Bank datastore, or BankDS, stores data in large, unordered arrays. That is, the BankDS allocates a large chunk of memory as an array, which it then populates as data comes in. When the allocated memory fills up, a new chunk of the same size is allocated, and storage continues. Internally, this is implemented as an array of arrays, where the outer array is an array of pointers to the inner arrays. When the outer array is filled, a new, larger array is allocated, and the contents of the first are copied to the second. By default, a BankDS inner array is large enough to hold 100,000

items. A larger size could improve performance by reducing the number of memory allocations, since allocations can require an operating mode switch and use up excess computing time; however, this can lead to wasted space if only a small portion of the array is needed. In the context of packet data, a new inner array would need to be allocated just 4 times per second at saturated gigabit speeds.

When an item is deleted from the BankDS, the location of the freed item is tracked in a last-in, first out (LIFO) stack, and when the next item is inserted, any freed locations are used before new locations are allocated. This avoids the costly process of having to compress the BankDS (and update all associated index references) on every deletion. Essentially, by replacing the hole with a new element rather than moving an existing element, BankDS tries to avoid having to do work more than once. Since BankDS has to keep track of deleted elements and their location until those spots are filled, this could cause memory and performance issues in a situation where deletions significantly outpace insertions; obviously, a situation where deletions are much more frequent than insertions is not sustainable, as BankDS will quickly run out of elements to delete.

Overall, the BankDS performs well. Because it allocates memory in large chunks, unnecessary system calls are avoided. However, BankDS performance can suffer if there are many deletions and it does not currently support raw iteration if there have been any deletions. The algorithm becomes significantly more complex and much slower, depending on the number of deleted elements, because the list of deleted elements must be checked each time the iterator is advanced in order to skip over deleted elements. This could be somewhat ameliorated by using a sorted data structure to store the location of deleted elements, or perhaps some sort of bitmap. In addition, as discussed above, BankDS does not return memory used by deleted items.

Once BankDS claims memory, it will always consume that memory, which can again cause problems in situations consisting of many deletions. A cleanup algorithm exists that can condense a BankDS, filling holes created by deleted elements and returning memory to the operating system if an inner array is vacated, but to avoid a loss in performance, the cleanup algorithm only runs under low-memory conditions; that is, if the system is about to run out of memory.

Lastly, the BankDS does not directly support the addition of elements of variable size. The size of memory allocations is fixed at creation time and cannot be modified. However, elements of size smaller than the fixed data size can be added, but the extra memory will be wasted. This can be somewhat mollified by using the indirect implementation of the BankDS, as the pointer stored can point to a variable chunk of memory. By tracking the size of each element in the arrays, BankDS could support the insertion of variable sized data, but this would come at the cost of both memory efficiency and performance, as well as complicate the design of the BankDS. In the context of traffic analysis, packet sizes, and more importantly, packet header-sizes are well-defined, a maximum of 1500 bytes in the former case, limiting potential waste.

Linked List Datastore

The other datastore currently included in ODB is the Linked List Datastore, or LLDS. As its name implies, it is implemented by a single-linked linear linked list. LLDS was designed for use when the limitations of BankDS are considered too onerous.

The LLDS does not perform as well as BankDS in most situations, because memory is allocated on demand, and incurs a slightly larger memory overhead than BankDS, as it must keep track of pointers from one element to the next. However, it does address some of the problems of BankDS.

For instance, the LLDS immediately returns the memory used by deleted elements to the operating system, unlike BankDS which never returns memory. However, random deletions are slow, requiring algorithmic complexity of O(n). Since the information is stored in an unordered fashion, additions to the LLDS have algorithmic complexity of O(1), as do deletions from the end of the list. The LLDS supports unordered iteration of elements, with no concern for deleted elements.

Most importantly, an extension of the LLDS allows for elements of varying size to be used; however, a small overhead penalty is incurred to keep track of the element's size. Despite this penalty, this extension, called the Linked List Variable Data Store or LLVDS, can greatly improve the datastore's memory efficiency when the variability in the size of stored items is large. This could be very useful if complete TCP flow payloads are stored, as there is neither an upper nor lower limit to their size.

3.1.2 Indexes

Access to the information in the datastores is generally done through the various indexes the user specifies. For example, a user could specify an index on TCP source ports, or another on destination IP addresses, allowing for quick retrieval of items based on those keys.

To create an index, the user must write a function (currently at compile-time) to specify a comparator, so that the index can sort the information in a logical fashion.

Indexes support flexible de-duplications, providing hooks allowing the user to write his or her own resolutions to duplicate keys. In the case of entropy-related algorithms, for example, these hooks allow for the creation of easily maintainable frequency maps.

Two different types of indexes are currently implemented: a tree index, and a linked list index, each with different advantages and disadvantages. The user should consider carefully what sort of use-case he or she anticipates when deciding on an index table.

Tree index

The first of the indexes is a tree-based index, currently implemented as a red-black tree. Like most trees, all operations are $O(\log n)$, including traversals. A small amount of overhead is used per element indexed, in the form of a pointer to the child nodes. A red-black tree is a type of self-balancing binary search tree; it was chosen because its self-balancing nature gives consistent performance.

Tree indexes form the backbone of most of the indexing under ODB. They perform equally well on all types of data, regardless of order. A user should probably be using a Tree index most of the time, particularly if keys are evenly-distributed.

Linked List Index

The second index-type available is based off of a simple linear linked list. Like the linked-list datastore, a single-linked list is used, but unlike the LLDS, data is ordered, based on the user-specified comparator.

The linked list index is well-suited to ordered insertions and accesses to the start of the list, where it outshines the tree-based index with a complexity of $O(1)$. However, random insertions, deletions and look-ups suffer under this design; they are $O(n)$. Though they have the same complexity, ordered traversals over a linked list index are generally more efficient than over a tree index.

In the context of analyzing network traffic, a linked list index is well-suited to indexing packets based on arrival times, and quickly accessing the X most recent packets.

3.1.3 Performance tuning

As discussed in Sections 1.1 and 2.3, insertion speed and memory efficiency are vital to the successful design of the system. Not only does the system have to be able to handle more than 300,000 packets per second, it must do so with minimal memory usage, in order to maximize the amount of data stored. Several steps were taken to maximize both computational performance and memory efficiency.

To give the system both thread-safety, flexibility, and scalability, a system of work queues was implemented. All tasks the system performs, such as insertion into an index tree, is written as an atomic (indivisible, as opposed to the ACID sense) work unit. These units are stored in a common work-queue, which the multitude of running worker threads access to obtain tasks. So when an item is added to a datastore, for instance, the associated index trees can be updated simultaneously in an asynchronous fashion as each update would be a separate work unit and would be assigned to a separate worker thread.

The entire system was designed using fine-grained, high-performance locks, and with the addition of work queues and atomic tasks, this allows the system to leverage the power of many-core machines. Fine-grain locking allows as many threads as possible to be operating on the system simultaneously. As currently designed, each index and datastore has its own separate locking mechanism, meaning that each can be worked on independently of the others. In testing, the system has shown almost linear speedup with additional threads of execution, the ideal for multithreading,

though returns tend to start diminishing around 4 or 5 threads of execution. The more complex the system of datastores and indexes, the more it tends to benefit from the addition of computing cores. Experience has indicated that the ideal number of threads is one more than the number of execution cores on the PC, or one more than the number of indexes (whichever is less), but this may vary based on workload and hardware.

The system also makes use of high-performance libraries developed by Google and others, notably tcmalloc, part of gpertools [26], which provide high-performance memory allocation routines. Many users have obtained good results from these libraries: github moved its database server to tcmalloc and obtained a 30% increase in performance [27].

Google's performance libraries also include an implementation of their own spinlocks. Google's spinlocks generally outperformed the locks provided by the default pthread library, but were less consistent; when using Google's spinlocks, there is greater variance in the performance results. Further, the spinlocks in the library were not designed with general development in mind and required some modification to adapt them to ODB.

Table 3.1 compares the performance of ODB using the default memory allocator and locking mechanisms of glibc, the default library under Linux, to that of the tools provided as part of gperftools. The test is the same as the one that was performed in section 2.3. As can be seen, tcmalloc and spinlocks can provide anywhere from a 13% - 60% speedup compared to glibc. This boost in speed can be crucial to meeting the requirement of achieving gigabit speeds on commodity hardware.

	50k	100k	500k	1M	5M	10M
ODB + glibc	0.15	0.25	1.17	3.11	18.24	29.92
ODB + gperftools	0.12	0.21	1.04	2.15	11.37	24.97
Percent speedup	25	19	13	45	60	20

Table 3.1: Comparison of libraries: Time to insert (seconds)

3.2 ACID Compliance

ACID (Atomicity, Consistency, Isolation and Durability) [28, 29] refers to a set of properties for database systems that ensures that transactions are completed properly. This section details the 4 aspects of ACID, and how ODB meets or fails each aspect, and why this decision was made with the goals of ODB in mind.

In the context of databases, a transaction is one or more operations, insertions, deletions, modifications, etc., that are grouped into a single event and are to be treated as a single action. ODB does not currently support transactions that are coarser grained than a single insertion or deletion and the associated updates to the index tables. Transactions become more relevant to systems that require performing a group of actions in an atomic, or all-or-nothing fashion, and systems that expect a multiplicity of simultaneous users. Since ODB was designed with cataloguing packet and network data in a highly efficient fashion in mind, most of the requirements are not relevant to the design.

Atomicity (in the context of database transactions) refers to a transaction's execution being all-or-nothing. That is, either the entire group of actions completes successfully, or none of them do. If some of the actions successfully complete, but a later action of the same transaction fails, all previous actions are "rolled back" or aborted. Since ODB does not support transactions coarser than a single insert, this is less of an issue — either the item is inserted into the datas-

tore, or it is not. While it is possible that an item could be successfully inserted into a datastore but the update to the index fails, this could only occur in an out-of-memory situation; the system at that point would have significantly more pressing issues to deal with. In theory, with proper pruning and memory reclamation routines defined, the system will never enter a low-memory state.

Consistency means that the system will only transition from one valid state to another; more specifically, it will never be left in an invalid state. As outlined above, it is possible for an item to be inserted into an ODB datastore but not the index tables, and while a missing index entry would be incorrect, it would not be invalid. The system would continue to operate without difficulty or the threat of failure. It is not possible for an item to be indexed without first being stored, a situation which could cause problems.

Isolation requires that the state of the system not be meaningfully affected by performing the actions or sub-actions in a concurrent fashion. That is, if the steps are performed out-of-order due to concurrency or scheduling, the system will not be visibly different to the user than if the same steps had been performed in an exact order or sequentially. This is relevant to the ODB system due to the high level of concurrent design incorporated in the system. While it is possible that two concomitant actions could be completed in a different order than when they arrived, this would have no meaningful effect on the system as a whole. In the worst case, the item inserted would end up in a different memory location than it otherwise would have or making the structure or order of the index tree slightly different.

Durability refers the persistence of data despite shutdowns, power offs, hardware failures, etc. Since ODB is strictly an in-memory database, it offers zero dura-

bility. Any attempt at adding durability could only come at the expense of performance, since it fundamentally requires accessing some form of permanent storage, such as a hard disk, which is orders of magnitude slower than using system memory. A lack of durability of data may actually provide an advantage in a security or privacy context. By not maintaining records or copies of data, many privacy concerns are greatly reduced. Further, since ODB is primarily interested in live captures, maintaining long-term records of traffic is not necessary. If long-term storage is still desired, a packet capture can be stored as a pcap file on disk, and read back into the ODB system when necessary.

While ACID compliance was not a goal of the ODB design, ODB partially meets the Atomicity and Consistency conditions, fully meets the Isolation requirement, but currently fails to meet Durability. If transactions larger than a single item are implemented going forward, more attention will need to be paid to these requirements, but for the purposes of this research, these limitations are not considered significant.

3.3 Limitations

The largest limitation of the system, as implemented, is its lack of flexibility. To get the high performance required of the design, it is currently necessary to define indexes and queries at compile-time. This prevents a curious user from running queries dynamically on data that is already in the system. Since the system was designed with the testing and prototyping of automated detection algorithms, rather than as a general database, the inability to run dynamic queries is not considered a significant drawback.

The other major drawback is the lack of durability of data. Since the data is

stored solely in memory, every time the system shuts down, all the data and the associated indexes are lost. In the case of forensics, or if research is being performed on an already-acquired or standard set of data, this drawback is minimized.

For the scope of this project, these limitations were considered acceptable trade-offs; the goal in mind is live capture and analysis of data, so a lack of long-term storage is not a significant concern.

3.4 Possible Deployment

Though ODB is designed with rapid prototyping in mind, it is possible that it could end up in commercial deployments. Its high-speed indexing of traffic-data, effectiveness on low-cost commodity hardware and high-memory efficiency make it suitable for use by both small and large firms.

After a particular algorithm has been designed and tested on ODB, said algorithm could be deployed with ODB and minimal modification to a production environment. Since the algorithm is working at desired speeds on ODB, it might make sense to deploy it as-is, instead of adapting the algorithm to a new situation.

In this case, it is envisioned that ODB and any associated algorithms or tools would run on their own separate host, virtual or physical, to reduce risks of ODB consuming the system memory and leaving little for another process, or vice versa. This host would be running some form of UNIX or Linux, and would either receive a dump of a network tap directly, or have access to some sort of logging store.

Multiple algorithms could be run on ODB without much difficulty, reducing the need for extra hardware and allowing for algorithms to potentially share work and resources.

The information provided by ODB and the algorithms running on it could serve to compliment a larger IDS system. However, as designed, ODB does not constitute an intrusion detection system. There is no provision in the system for monitoring and/or alerting administrators; however, such a provision could be added if and when the system is deployed.

3.5 Future work

As implemented, the system is useful for the collection of network traffic and the evaluation of intrusion detection algorithms. However, there are several areas the designers of the system would like to see improved upon in the future.

Disk-backed datastore A new type of datastore could be designed that keeps values in memory initially, but backs them to disk, either on demand when more memory is needed, or continuously, taking advantage of lulls in insertions for the disk I/O to catch up. Additionally, the data could be read from the disk and the system populated on start up; this would enable the system to provide the lacking 'Durability' of ACID compliance.

Flexibility Currently the system requires that the user specify his or her indexes and comparators at compile-time with no provision for running queries dynamically. With the recent addition of atomic tasks, it is possible to write some sort of front-end shell that allows a user to dynamically load and execute code on a running system.

Hashmap Index A new index type could be created which indexes items through the use of a hashtable instead of a tree or linked list. This has some advantages,

such as O(1) insertions, deletions, and look-ups, while minimizing overhead to just one pointer per item, instead of the two required by both the Linked List index and the Tree index. However, depending on the design, there could be some wasted space in the form of unused hash locations, or performance could be reduced by having to occasionally grow the hashmap. Also, a hashtable precludes the ability to perform ordered traversals.

Compressed Datastore It might be possible to gain additional memory efficiency by performing compression on the elements in the datastore. Naturally, this would come at the expense of significant insertion and look-up speed, but would probably still be faster than backing items onto a hard disk. There would be some minor gains to be made in the case of packet header data, as well as plain-text payloads, but binary payloads, such as media files and images, tend to already be heavily compressed, negating much of the advantages of using a compressed datastore.

Clustering The system was designed to deal with data from a single 1Gbps link, and while it achieves performance sufficient to handle the data from such a setup, increasing the network speed to the next standard step, 10Gbps, necessitates the use of clustering, or some form of distributing computation across multiple nodes. It is simply not possible for a single host, on today's hardware, to perform analysis on a full packet capture from a saturated 10Gbps link; there is too much data. With the addition of a distributed computation interface, the ODB system could begin to scale across multiple hosts, and potentially could be used to analyze traffic on even larger, faster networks by using the extra computing power and memory available on a cluster.

Chapter 4

Data Sources

In order to test a potential anomaly detection algorithm, there is a need for a good dataset with known anomalies in it to test the candidate algorithm. This chapter reviews existing data sources, dissects why they are unsuitable, and discusses the construction of a new dataset.

4.1 Unsuitable Data Sources

4.1.1 DARPA Datasets

In 1998 and again in 1999, the Defense Advanced Research Projects Agency (DARPA) held an anomaly detection competition. For this competition, DARPA created a standardized dataset, called IDEVAL, (commonly referred to as the 'DARPA dataset'). These datasets simulate the behaviour of a large network with many hosts over the period of several weeks. Each week, several anomalies appear on the network in the form of outside attacks on the hosts on the network.

Age is one of the main problem with the DARPA datasets. They are more than 13

years old at the time of writing: an eternity in the world of computers and computer networks. Since that time, the nature of network attacks and anomalies as well as basic network usage and traffic have changed significantly. The DARPA datasets employ out-dated and obsolete protocols and make use of attacks that may have been common at the time, but are now obscure and are comparatively primitive. For instance, the fdformat attack, a mainstay of the anomalies in the DARPA dataset, attempts to exploit a vulnerability in UNIX-like operating systems that has been fixed since 1997.

To put things in perspective, Windows XP, still the most popular current operating system and subsequently attack vector, despite being 11 years old, was more than 2 years away from release at the time the DARPA datasets were created.

A study by Mahoney and Chan [30] identifies potentially unintentional artifacts in the 1999 DARPA dataset which could allow potential detection algorithms to identify intrusions into the network that would otherwise be missed. These unintentional artifacts are the result of idiosyncrasies of the simulation's underlying implementation, and essentially make things too easy for any potential detection algorithm, which can lead to suspiciously good performance of mediocre algorithms. Examples include similarities of TCP SYN packets, predictability of source addresses, and errors in the packet checksums. The researchers even tested intrusion detection algorithms using the DARPA dataset against a capture that included a mixture of both DARPA data and real-world data, finding that the algorithms performed better against just the DARPA dataset. This finding indicates that the artifacts in the DARPA set were enabling the algorithms to find anomalies that they would otherwise miss.

In "Testing Intrustion Detection Systems" [31], John McHugh discusses shortcomings of intrusion detection algorithms developed by DARPA and Lincoln Labora-

tories. Several of his criticisms focus on the nature of the datasets used, in this case, the DARPA 98 and 99 datasets. He notes that the measures used to make simulated traffic similar to real world traffic are not supplied. He also notes that the data rates are suspiciously low for a network that supposedly contains hundreds of users and workstations. According to [31], the average rate of the DARPA dataset is on the order of 10s of kilobits per second, but a similarly sized real-world network generated over 500 Kbps.

In attempting to use the DARPA dataset for this thesis, it was found to be too sparse, and exhibited odd periods of little to no traffic. Hours would pass in the dataset without a single TCP packet being generated. This behaviour runs counter to both intuition and experience, further eroding confidence in the DARPA dataset's suitability.

4.1.2 Other Existing Data Sources

While other datasets exist and are used for statistical analysis, these datasets are either unavailable or unpublished because of non-disclosure agreements (NDAs), or are unsuitable for use. This is a list of some of the datasets that have been used by other researchers, but were, for a variety of reasons, either unavailable or unsuitable for use.

GEANT - Unavailable due to NDA. Also, significantly old (9 years).

Abilene - Only metadata available. Also, significantly old (10+ years).

Sprint-1 and -2 - Unavailable. Also, significantly old (9 years).

4.2 Construction of Data Source

To remedy this lack of a good, standard dataset against which the efficacy of intrusion detection algorithms could be measured, MERLIN provided assistance with producing a new dataset with known anomalies.

As an Internet Service Provider for Manitoba's schools and education providers, MERLIN manages the network access for tens of thousands of computers and users. They have recently had to expand beyond gigabit capability to keep up with the growing demands of their clients.

MERLIN graciously allowed the capture of a week's worth of data from one of their largest clients, Brandon University. Unfortunately, and quite understandably, many of the more aggressive attacks could not be run on MERLIN's hosts directly, as they did not want to perform a denial-of-service on an actual, real-world web server that was serving requests to clients. In order to run these more aggressive attacks, a smaller network segment was built, consisting of about 15 hosts running on a virtual machine. This segment was put in close proximity to the rest of MERLIN's network, and all traffic to and from this segment went over MERLIN's actual infrastructure. This would allow for the more aggressive intrusion attempts without damaging MERLIN's clients while maximizing the reality of the simulation. A diagram describing the high-level layout of MERLIN's network and the inclusion of attack and target hosts can be seen in Figure 4.1.

This network segment was hosted on an Intel Xeon E5440 running Citrix XenServer 6.0.2. It contained 4 Windows XP SP1, 7 Windows 7 64bit and 2 Debian Linux hosts. This machine also hosted the attack boxes, 3 Debian Linux hosts, which were routed through a separate interface.

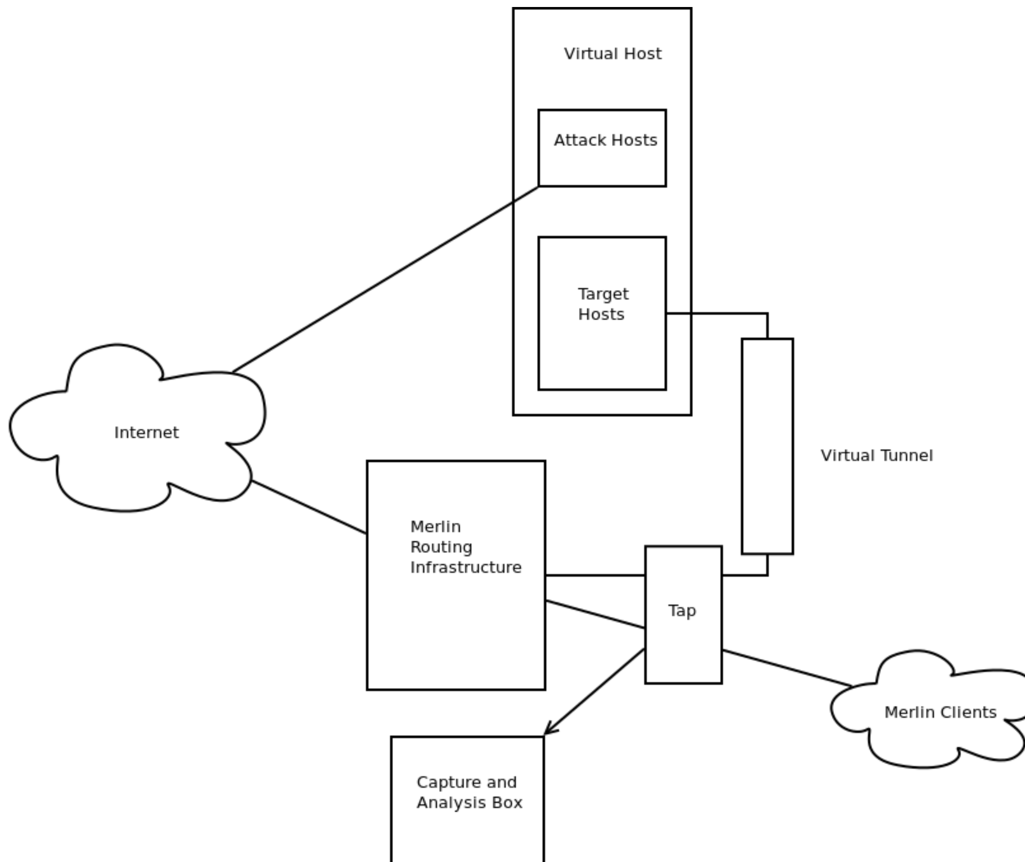


Figure 4.1: Network Diagram

The biggest advantage to this approach is that it gives access to large quantities of real world data, the sort of network traffic that computers generate everyday, with reduced risk of artifacts as in the DARPA datasets. Further, because of MERLIN's use of firewalls and other intrusion detection and prevention software, outside anomalies (i.e. ones not injected by the attack network) could be reduced to a minimal level. This also allowed for the injection of very specific attacks, with known behaviour and parameters.

4.2.1 Injected Attacks

The attacks injected were chosen for their commonality, simplicity, and potential visibility to the chosen algorithm. This meant that all attacks had to be TCP-based, omitting some of the more common ICMP-based scans and Denial of Service (DoS) attacks, such as ping scans and floods, and any DNS-based attacks, such as spoofing, scanning, and tunnelling. Only relatively noisy attacks were chosen — that is, attacks that generate a substantial amount of traffic — to avoid attacks that might attempt to fly in under the radar and escape detection, since the algorithm under test is more interested in significant shifts in traffic patterns. The pattern and distribution of the attacks was chosen by a random number generator, python 2.6.6's random module with a specified seed, 1, guaranteeing replicatability.

Start times were chosen by a function, such that no two attacks could start in the same 15-minute window, though long-running attacks could overlap. The attack would last for a time chosen at creation such that the majority of runs would be short lived, but there would be the occasional long-term attack that could span hours. The function for attack duration to provide this distribution used was:

$$\frac{M}{2^{xS}} + m$$

Where x is a random number between 0 and 1, S is the skew rate (chosen to be 5), M is the maximum run time, in seconds (chosen to be 3 hours), and m is the minimum run time (chosen to be 5 minutes). These values are somewhat arbirtary and where chosen to fit the desired distribution as discussed above. For the scope of this thesis, the particular distribution of attack duration is not a significant concern, provided that the above conditions are met.

Denial of Service (DoS) Two kinds of DoS attacks were simulated. A standard denial of service attack by flooding a service with TCP SYN packets and a more sophisticated "slowloris" attack, which ties up resources on a web server. For the first sort, the open-source tool mausezahn [32] (or mz, version 0.40) was used. Mausezahn (German for "mousetooth") floods a connection with TCP SYN packets by writing directly to the network interface. One hundred thousand (100,000) TCP SYN packets were sent to either the ssh or Apache web services on one of the Linux targets, with randomized source IPs and source ports.

The second type of DoS used, the slowloris attack (named after a species of small primates) [33, 34] ties up web server resources by making many very slow HTTP requests simultaneously. Most web servers limit the number of active threads and/or connections the service can use, ironically to thwart other kinds of DoS attacks. The request is kept alive by sending data very slowly, at the rate of a few bytes a second, so the connection never times out. Since a slowloris attack never (or very gradually) completes the HTTP request, the web server can never free the resources used by connection. Many web servers are vulnerable to a slowloris DoS attack by default. Pyloris' httploris script (version 3.2) [35] was used to perform the attack, with a delay of 0.1 seconds between both spawned connections and threads. Naturally, neither type of DoS attack was used on MERLIN's client network; both were targeted solely on the private hosted target network.

Because of the amount of traffic generated, the basic DoS attack is highly visible, and is usually easily identified by the most basic of intrusion detection systems. The second, however, is quite stealthy, since it can be implemented in the space of only a few hundred packets — even the most advanced IDSs can have a

hard time detecting slowloris attacks. Between these two attacks, we have what amounts to an easy and a hard to catch DoS.

SSH Bruteforce A very common attack, upon discovery of a running Secure Shell (ssh) service, an attacker will often attempt to bruteforce access to the host, using common usernames and a dictionary of passwords. This was simulated using the brutessh python script (version 0.2, part of the edgessh suite) [36], and a small dictionary of about 3000 passwords. The attacking box attempted to login as the root user on one of the Debian Linux target hosts, but the password to login was not in the dictionary file, and so the attack never succeeded. The default number of concurrent ssh threads, 12, was used.

Though SSH bruteforcing is usually detected by analyzing ssh logs, by including a statistical analysis of network traffic, additional credibility and correlation of the attack can be performed. The addition of statistical analysis may allow the attack to be detected sooner, possibly preventing intrusion.

Host Scan This attack is a type of reconnaissance attack. In preparation of a more specific intrusion, an attacker may attempt to enumerate all the hosts on the network. This can be accomplished in numerous ways, for instance by sending an ICMP echo request to all the IPs on a network and listening for replies, or likewise with TCP SYN packets. This was simulated by using the popular tool nmap (version 5.00), which sent a single TCP SYN packet on port 80 to various chunks of the scannable network space.

Detecting this sort of attack, and other scans, requires a bit more sophistication on the part of an IDS. It's not enough to simply detect 1000 SYN packets sent to a single port, or even to watch for packets sent to successive ports, advanced scanning tools are capable of breaking up requests both over time and over large

parts of the network, and avoiding detection.

Service Scan After discovering a host, an attacker will often attempt to paint a detailed 'picture' of the target, by scanning it for running services, and attempting to discover the OS version through fingerprinting. This behaviour was simulated through three distinct attacks, all implemented using nmap.

In the first case, if a host was detected (using the method as described in Host Scan, above), the first 1024 ports of the host were scanned for listening sockets.

In the second case, all 1024 ports of all hosts in a chunk of the scannable space were scanned, regardless of their being detected or not.

Lastly, a detailed scan was attempted of the ssh service of all hosts in the specified range.

For a detailed listing of which attacks were performed by which host, as well as their start and end times, please refer to Appendix A.

4.3 Dataset Shortcomings

Though the approach overcomes many of the faults of the dataset provided by DARPA, the approach used to construct this dataset has a few shortcomings.

For instance, the target portion of the network, used for receiving the more intrusive attacks, was simulated on a Xen virtual host. Ideally, real hosts would be used instead, to guarantee maximum authenticity. This is not a significant drawback, though, as the differences between the sort of traffic generated by a hosted machine compared to a physical host are likely to be minimal. Furthermore, hosting web and

other services on virtualized hosts is becoming an increasingly common practice on many networks.

Additionally, the target portion of the network was on a physically separate network from the real portion of the network. This could cause differences in traffic that some learning algorithms may latch onto, skewing results. Differences in traffic were minimized by forcing the traffic to the target network to follow as much of the same path as that of the whole network and choosing IP spaces and ASNs that matched that of the network as closely as possible.

Lastly, as the dataset was only generated from a portion of MERLIN's network, the dataset was not generated at gigabit speeds, averaging instead only 10.5Mbps over the week-long period. Though the system is capable of handling packets at a rate greater than a gigabit network could generate them, as can be seen in section 6.2.2, algorithms that do work periodically may show greater throughput than they would actually be capable of when using this dataset for testing.

Chapter 5

Entropy as Anomaly Detection

This chapter discusses the algorithm chosen to test the capabilities of ODB, providing some background to the algorithm and foundational theory.

5.1 Entropy of Network Traffic

Using statistical measures to detect anomalous network behaviour is not a recent innovation. Administrators have for years relied on simple counts of SYNs to FINs, corrupt packets, and others to determine the health of their networks [37].

Entropy is a second-order statistical measure, that measures the 'randomness' or 'disorder' of a distribution. Recent research has shown that in normal operation, a computer network has a particular distribution or fingerprint, which can be described by measuring the network's entropy [38].

Even though there is a great deal of variance in network traffic from moment to moment or even day to day, the level of randomness remains consistent. A certain proportion of ssh, DNS and web traffic is generated, with some sites, like Facebook or Google, being more visited than others. Variations in this fingerprint can be a good

indication that something on the network has gone amiss, such as a viral infection, or the compromising of one or more hosts on the network by an intruder.

5.2 Principal Component Analysis

When analyzing network traffic, there is not a single statistic, but many statistics that need to be compared and analyzed to determine the health of the network. Source and destination addresses, packet length, originating ASN, ports used, flags set, and even the bytes of the payload itself can all be used to measure the behaviour of the network, each one of these having a greater or lesser degree of correlation with the others. Further, by capturing traffic at multiple points on the network, the feature set is further increased. The result is a set of data that is highly dimensional and hard to decipher in a meaningful way. In addition, the variables available for analysis may vary from application to application, or even from deployment to deployment — a particular client may not be able to expose TCP window sizes, for example — requiring additional work to tune the algorithm to each situation.

One method of reducing the dimensionality of data, while helping to make algorithms more generic, is Principal Component Analysis (PCA). PCA maps data onto new orthogonal dimensions, or components, such that the new dimensions are chosen in order of decreasing variance. That is, PCA chooses the first dimension to have maximum variance; the second dimension is chosen to be orthogonal to the first, such that it also maximizes variance. This process is continued until a new complete set of dimensions is chosen and the last dimension expresses the lowest variance of the data.

In addition to ordering dimensions in terms of variance, the new set of components

are linearly uncorrelated. Further, it loses its specificity; for example, instead of representing “entropy of window size”, the new dimension is a generic measure. This makes algorithm implementation simpler across mutliple configurations.

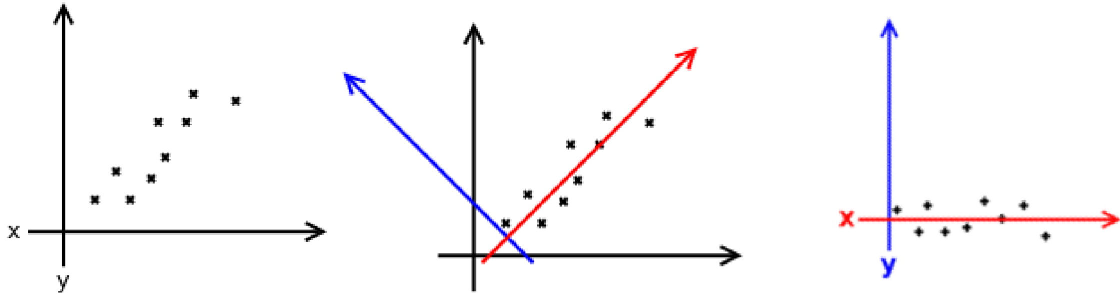


Figure 5.1: Principal Component Analysis

In Figure 5.1, the first picture shows a set of data plotted along two axis. You can see that there is a great deal of correlation between the x and y values.

The second diagram shows the two new dimensions as chosen by PCA. The first dimension, shown in red, was chosen to capture maximum variance, and as this simple example is only in 2D space, the remaining dimension, in blue, was the only one orthogonal to the first. The third diagram shows no correlation between the data, plotted along the two new dimensions, $\bar{x}$ and $\bar{y}$.

In terms of information entropy, with the assumption that a channel, $\mathbf{x}$ can be described as $\mathbf{x} = \mathbf{s} + \mathbf{n}$, where $\mathbf{s}$ is the signal and $\mathbf{n}$ is the noise component, PCA has been shown to be optimal for reducing the dimensionality of the data stream [39].

In a series of papers, Lakhina et al. use this property of PCA and information theory to separate the ‘signal,’ or normal traffic, from the ‘noise,’ or abnormal traffic into two separate measures [7, 8, 9, 10].

Lakhina et al. sampled statistical traffic information from three large backbone networks, Sprint, Abilene, and Geant. These large networks contain few hosts, but

instead pass large quantities of data between endpoint networks. Each packet of data enters the network at a particular point of presence (PoP), and exits via another, forming an origin-destination (OD) pair for that packet. Lakhina et al. could measure the entropy of IP 4-tuple (source and destination port and IP) for each OD pair over time. This could result in a set of data with a very large number of dimensions, over 300 in some cases, as there were 4 'views' of entropy for each OD pair, and the largest network, Sprint, had 78 OD pairs.

According to Lakhina et al., normal variation in traffic flows occupies a relatively low dimensional space [10], so knowing this, they found that they could break up their traffic measurements $\mathbf{x}$ into normal and anomalous subspaces, $\hat{\mathbf{x}}$, and $\bar{\mathbf{x}}$ respectively, such that $\mathbf{x} = \hat{\mathbf{x}} + \bar{\mathbf{x}}$, as above. This way, by using PCA, they could draw out the high-variance dimensions and map them to the normal space, thereby relegating all remaining dimensions to the lower variance anomalous space.

Under normal conditions using this method, the magnitude of the data in the anomalous space would be low. By calculating the squared prediction error (SPE) of the anomalous subspace, which is $||\bar{\mathbf{x}}||^2$ for each time bin, and watching for values above a set threshold value, α, Lakhina et al. found they could detect the majority of anomalous features while providing a very low false alarm rate.

5.3 Algorithm Evaluation

While Lakhina et al. obtain good results using PCA with entropy measurements of OD flow data on very large networks, they do not address the applicability of their algorithm to smaller networks desiring real-time or near real-time anomaly detection.

This section discusses the differences between the approach used by Lakhina et

al. and those used by this thesis.

First, due to their access to a much larger dataset, Lakhina et al.'s data had much higher dimensionality than that produced by this dataset. While Sprint has 13 PoPs, the dataset used in this thesis was constructed on a network of essentially one. To mitigate this, entropy calculations were performed on more traffic features. While Lakhina et al. only looked at source and destination port and IPs, the applied algorithm also utilized sequence and acknowledgement numbers, TCP flags, TCP window sizes, and payload lengths.

Second, Lakhina et al. analyzed data on OD-flow data, whereas this thesis analyzes traffic features at the packet level. A flow contains the metadata from the start of a TCP-session to an end: duration, origin and destination, but little-to-no information about the individual packets themselves. This also biases the capture to give smaller, shorter sessions equal weight as that of long, drawn-out sessions that transport a great deal of traffic.

Next, Lakhina et al. used sampling on the order of about 1 in 1000 flows. This thesis works on a full capture.

Lastly, Lakhina et al. gave no indication of the relative performance of their algorithm. In the context of this thesis, the ability to analyze network traffic at line speed or greater is one of the main objectives, and so the algorithm was implemented with performance and evaluation thereof in mind.

5.4 Implementing Algorithm on ODB

5.4.1 Advantages of ODB

Several features of ODB make it well suited to performing the calculations and analysis of the adapted algorithm.

Entropy calculations require keeping track of frequency maps. That is, it is necessary know how many times each value appears in the dataset. ODB's powerful de-duplication options, as discussed in section 3.1.2, allowed for easy merging of duplicate values, while keeping track of their counts and minimizing wasted memory and processing time.

Once the data has been collected, the algorithm requires repeated ordered traversals of the dataset. Again, ODB supports fast iterations of both indexes and the datastore itself, making it well-suited to this particular algorithm.

5.4.2 Workflow

This section describes the workflow performed by the system in order to execute the algorithm as described in section 5.2. The workflow is outlined in the flowchart in Figure 5.2 and described in detail below.

After initializing the datastore and indexes, the program starts by processing packets from the specified packet capture file, or standard input. The program reads in packets and adds them to the datastore and indexes, until the timestamps on the packets indicate that a window, or time bin, has expired. This signals that it is time to perform the first iteration of entropy calculations. When a packet is added, its metadata (source and destination IP and port, packet length, etc) is parsed by

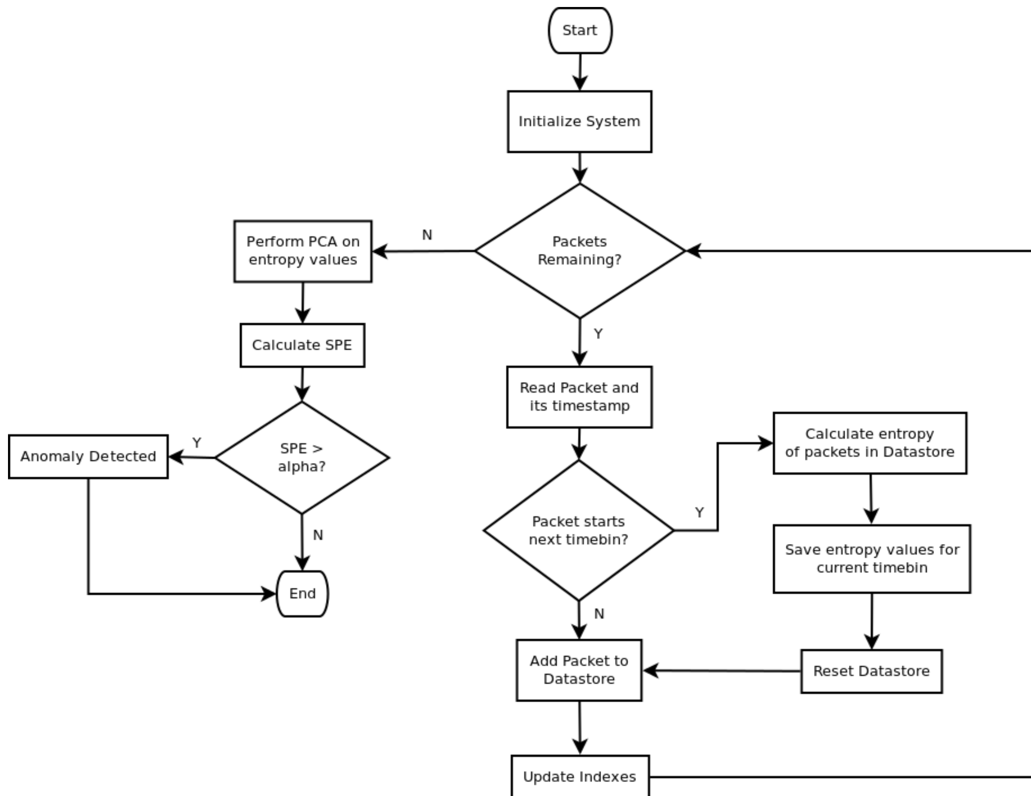


Figure 5.2: Algorithm Flowchart

the ODB instance, and indexed in the appropriate tree. Duplicates are dropped, saving both memory and processing time, but counts of duplicates are maintained for analysis at the completion of the current time bin, in this case, every 5 minutes.

Once the analysis is started, reading is paused while the calculations are made. With additional work, reading in could continue in a new, separate datastore, potentially yielding additional performance, but this was not implemented.

Analysis starts by building a frequency map of values, and calculating their entropy values. The entropy values are stored for later use, and the reading of data continues until the next time bin is complete.

Once all the data has been read, analysis progresses to the next step. The entropy values are normalized against the maximum entropy value, and added to a matrix.

Normalization is performed, as per Lakhina et al., to prevent one particularly entropic measure from dominating the results, translating values onto a linear scale from $[0, 1]$. This matrix is passed off to the PCA algorithm, and PCA is performed. The PCA calculations were adapted from Michael Niedermayer's open-source (LGPL) implementation [40]. The results are then written to standard output.

The last step is performed by a simple Python script, which takes these final values, graphs them, and calculates the squared prediction error (SPE), $||\bar{x}||^2$. This script both generates graphs and identifies points in time where the SPE exceeds the threshold value, α.

Chapter 6

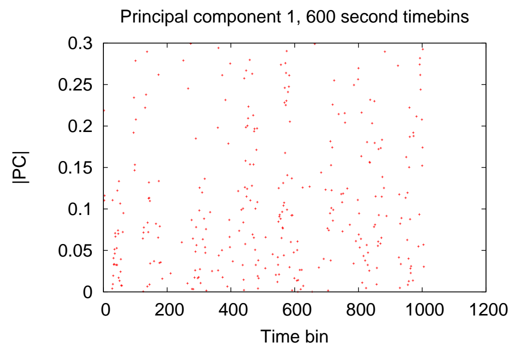
Results

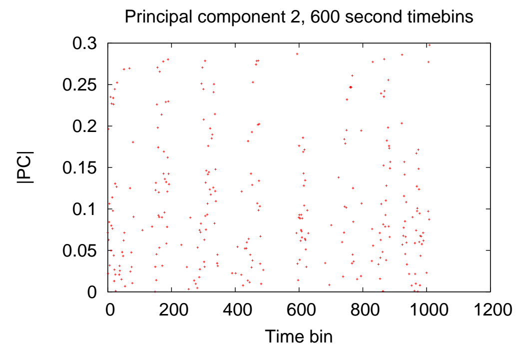
This chapter discusses the results of using ODB to implement the algorithm as designed by Lakhina et al. First, the algorithm will be tested against the created dataset, and then the system itself will be evaluated against the criteria laid out in Section 1.1 and Chapter 3.

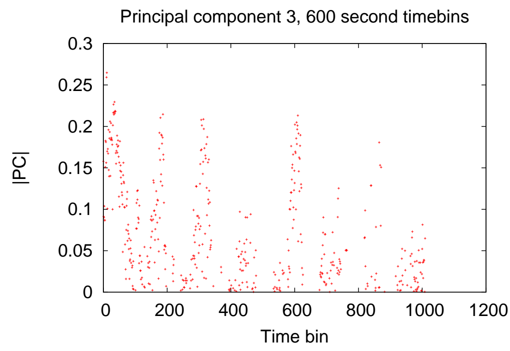
6.1 Evaluation of Candidate Algorithm

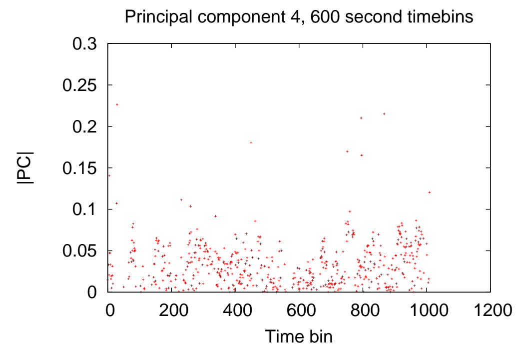
This section discusses the algorithm designed by Lakhina et al, as adapted to the dataset and ODB. An intrusion detection algorithm is generally measured in its ability to detect a high porportion of anomalies while returning few false positives. In their papers, Lakhina et al. were able to achieve an detection rate of nearly 100%, depending on the anomaly, with a false positive rate as low as 6%.

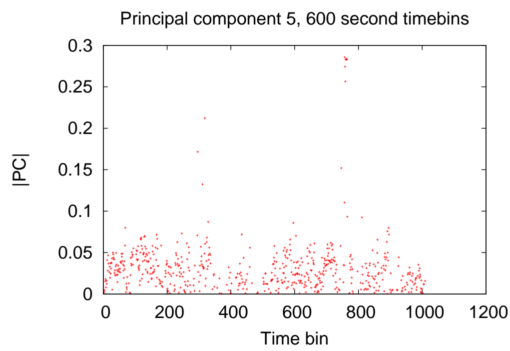
The images in Figure 6.0 on page 51 and 52 demonstrate the breakdown of network traffic data after performing principal component analysis. It can be seen that the first 3 or so principal components demonstrate a strong diurnal trend, rising and falling in tandem with daily usage patters. However, these tendencies are almost completely

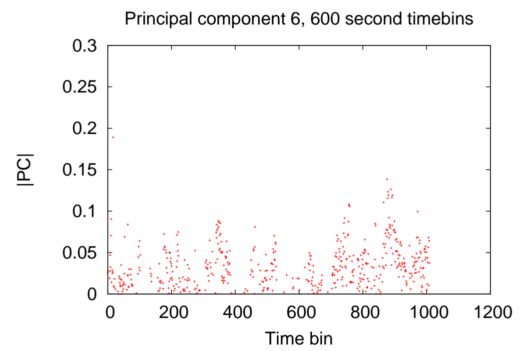

Principal component 1, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200


Principal component 2, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200


Principal component 3, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200


Principal component 4, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200


Principal component 5, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200


Principal component 6, 600 second timebins
|PC|
Time bin
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200

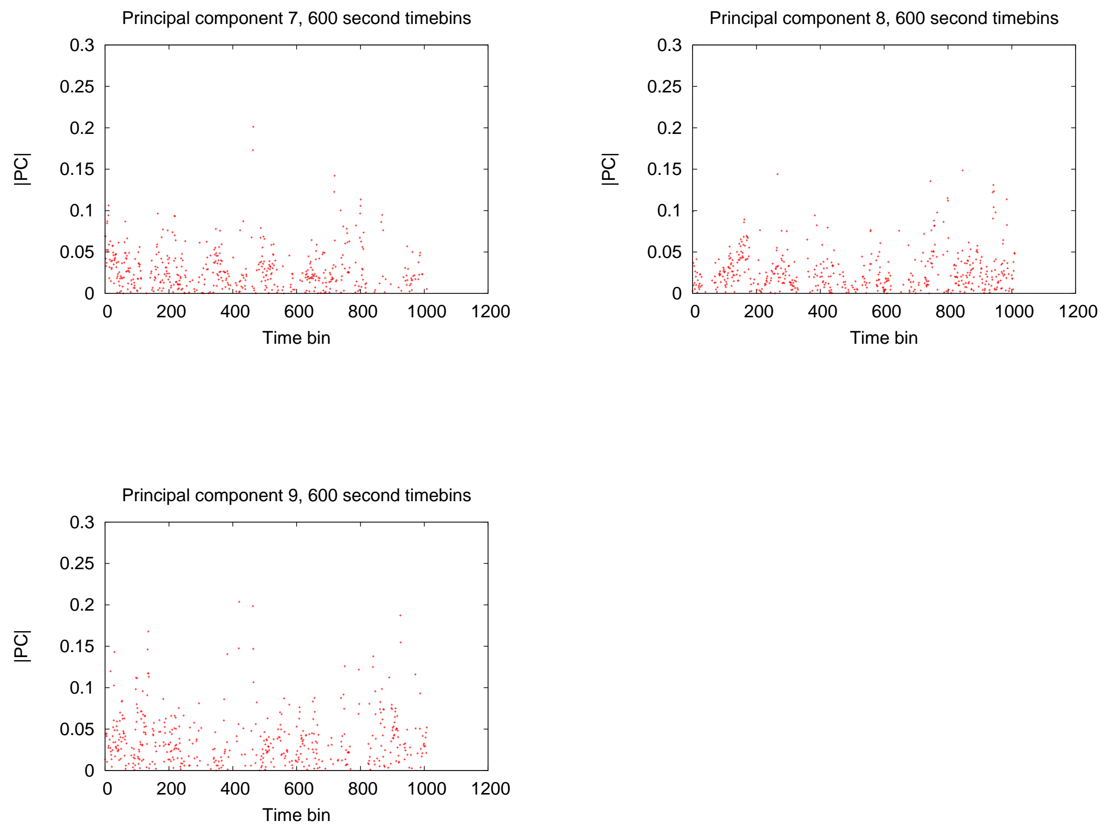


Figure 6.0: Principal Component Analysis of Normal plus Anomalous Traffic

gone by the last principal components, which to the eye are indistinguishable from noise.

The next two diagrams on page 54 demonstrate the effect on the squared prediction error of anomalous, hostile network traffic. The first graph, 6.1a, is the SPE of network traffic without the anomalies. The second graph shows the SPE of the same network traffic, but this including the injected anomalous traffic.

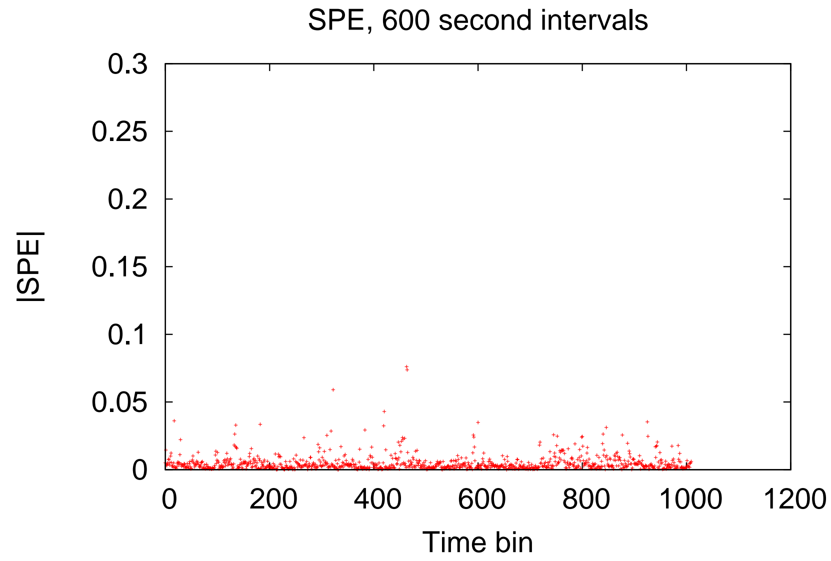
To the naked eye, several candidates immediately jump out for investigation. Using the analysis script, with a threshold value of $\alpha = 0.06$, these spikes were collected and are shown in Table 6.1. In this case, spikes above the threshold that continued across consecutive timebins were grouped into a single spike, the rationale being that consecutive spikes are probably caused not by a new anomaly, but by an existing, continuous one.

Unix time	Magnitude	Consecutive bins	Anomaly detected
1355043001	0.0708	1	none
1355050201	0.0776	1	nmap -PN -sS -p 1-1024 on atk2
1355223601	0.0693	1	none
1355284201	0.0849	2	none
1355310601	0.2925	3	nmap -PN -sS -p 1-1024 on atk2
1355317801	0.0844	1	none
1355321401	0.0796	1	none
1355487001	0.1030	3	none
1355489401	0.0999	3	none

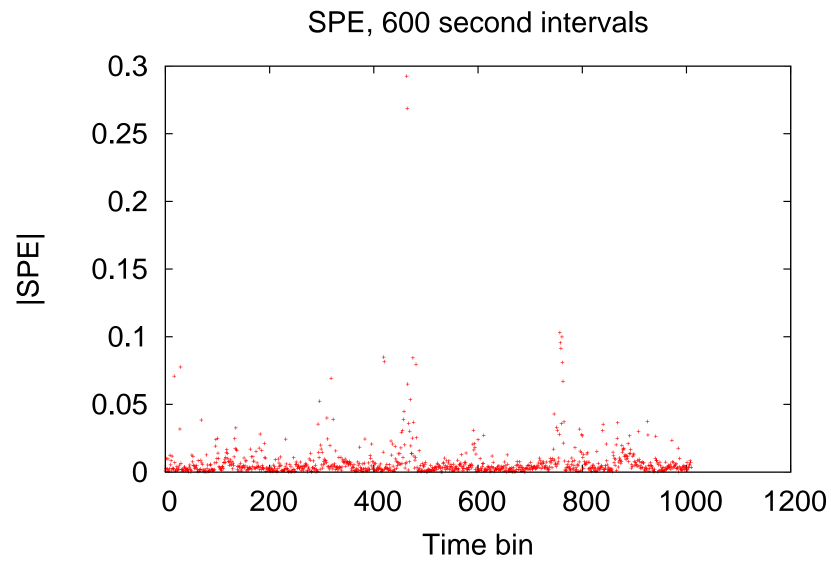
Table 6.1: Timebins with SPE exceeding threshold

A spike is considered to have detected an anomaly if the spike is in the same timebin as the start of the attack, or the timebin immediately following the start of an attack.

With more than 50 anomalies injected in the dataset, the algorithm was only able to detect two while returning 7 false positives at the threshold level of $\alpha = 0.6$.



(a) Normal traffic



(b) Normal plus anomalous traffic

Figure 6.1: Squared Prediction Errors (SPE)

Lowering the threshold level did little to improve detection rate, while increasing the number of false positives generated. An examination of the dataset at the time of the false positives also did not reveal any external anomalies originating from the general Internet.

It is worth noting that the two detected anomalies were of the same type: a service scan of the first 1024 ports of every IP on the network. The fact that this particular anomaly was detected twice might suggest that the algorithm is well-suited to detecting this particular type of anomaly, except that an additional anomaly of this type went undetected. The SPE for that timebin did not exceed 0.032, well below the threshold value.

Based off of these results, specifically a high false positive rate and low rate of detection, it would seem that the PCA and entropy-based algorithm as designed by Lakhina et al. is not well suited to smaller networks and/or analysis at the packet level, as opposed to the flow-data of larger backbone networks it was originally designed for.

6.2 Performance of ODB

This section evaluates the capabilities of ODB against the software requirements laid out in section 1.1 and chapter 3. To reiterate, those are:

1. The ability to quickly adapt new algorithms and methods for anomaly detection to a variety of similar datatypes
2. The ability to quickly store, index and process incoming data at gigabit speeds on commodity hardware

3. The ability to handle the large quantities of data generated by gigabit links

6.2.1 Ease of Use

Evaluating ease of use is always a subjective measure and difficult to quantify. In this case, the candidate algorithm was adapted to ODB in just over 500 lines of C code, excluding comments, whitespace and external libraries. Approximately half of that was the algorithm itself, the rest being code for processing and inserting the packet data into the database system. This suggests that the system is well-suited to quick adoption of new anomaly detection algorithms.

ODB is currently being used by another security researcher, Michael Himbeault, to detect DNS tunnels in network traffic, a similar but distinct use-case, demonstrating that ODB can be used for more than just PCA/entropy analysis on TCP traffic.

6.2.2 Processing Speed

One of the main design goals of ODB was to process incoming packet data quickly, and many design choices were made with this in mind. As was discussed in section 2.3, to perform at gigabit line speeds, such a system would need to process 300k-400k packets per second, or 10 million packets in about 25-30s.

Table 6.2 reproduces the results obtained in section 2.3 , but includes the performance of both ODB and ODB with the gperftools library. As can be seen, ODB performs much better than any of the traditional database systems, and even exceeds the lower-end of the performance requirement of 10 million packets in 30 seconds. With the inclusion of gperftools, the margin widens even further.

Database	50k	100k	500k	1M	5M	10M
MySQL	7.73	15.91	82.08	167.27	1055.39	2045.25
PostgreSQL	18.25	34.61	166.10	332.09	2874.71	11364.23
SQLite	1.00	2.08	17.86	119.25	12194.45	39982.88
SQLite (in-memory)	0.75	1.63	8.95	17.96	103.39	217.53
ODB	0.15	0.25	1.17	3.11	18.24	29.92
ODB (gperftools)	0.12	0.21	1.04	2.15	11.37	24.97

Table 6.2: Comparison of databases: Time to insert (seconds)

While performing analysis on the collected data, ODB plus the PCA/entropy algorithm could process 782GB of data in less than an hour. This is more than 217MB/s, almost twice as much as the 128MB/s a gigabit line is capable of producing.

It is worth noting that the data capture provided by MERLIN does not represent a full gigabit network. While ODB was able to process the data at better than gigabit speeds, the 782GB capture represented a full week of data. This represents an average throughput of only 10Mbps. While there was no gigabit network available for testing, these numbers suggest that ODB would have no problems processing data at that rate.

6.2.3 Memory Efficiency

The ability to handle large quantities of data is largely dependent on the system's memory efficiency. As described in chapter 3, the system was designed to minimize the amount of memory overhead per element. Depending on the choice of datastores and indexes, the system can use as little as 16 bytes per element stored of overhead per index, and a very small amortized amount on the datastore.

In this test, each system inserted the specified number of elements, each consisting of a 512 byte key and a 1024 byte value. The keys and values were both random

strings of upper-case letters, minimizing the chance of duplicates. The numbers that were recorded are the system's resident memory usage, as reported by the operating system, via the console tool htop. A 100% memory-efficient system, with no overhead, would expect to use $\frac{1000000(1024+512)}{2^{20}} = 1464$MB to store 1 million elements.

As can be seen from Table 6.3, only TokyoCabinet's tree storage engine could match ODB in memory efficiency. ODB's memory efficiency is very good, incurring a total overhead of about 2%. Naturally, since overhead is constant to the number of elements rather than their size, smaller elements would yield less memory efficiency, while larger ones would improve it.

Database	10k	50k	100k	500k	1M
Ideal	15000K	75000K	146M	732M	1464M
ODB	16232K	77740K	150M	748M	1496M
Redis	20268K	95616K	185M	921M	1840M
SQLite (in-memory)	59776K	178M	568M	2831M	-[1]
TokyoCabinet (hash map)	16936K	78712K	152M	756M	1511M
TokyoCabinet (tree)	16668K	77920K	150M	748M	1496M

Table 6.3: Comparison of in-memory databases: Memory usage (Bytes)

It is worth mentioning that Redis and TokyoCabinet are simple key-value stores, unable to create more complex systems of addressing and indexing, and thus rendering them somewhat unsuitable for use by an analysis algorithm, despite their relatively good memory efficiency.

[1] The host had insufficient memory to complete this test

Chapter 7

Conclusion

This thesis detailed the design, construction, and implementation of an in-memory datastore for the rapid development and evaluation of anomaly detection algorithms. To that end, an existing anomaly detection algorithm was selected and adapted for this system. Finally, a new dataset with known anomalies was created to evaluate said algorithm.

The motivation for creating such a system was to assist researchers with the design and evaluation of new anomaly detection algorithms, and to provide tools and a framework for rapidly deploying such an algorithm in a live environment. This meant that the system had to be both easy to use, but fast enough for live analysis and capture. In going with an in-memory solution, the system also had to be memory efficient enough to store and analyze large quantities of data.

Once designed, the system had to be tested, and for this purpose, a promising anomaly detection algorithm was chosen. Lakhina et al. had produced good results using an entropy-based algorithm on large, backbone networks, and this was selected and adapted for use on a smaller network, to demonstrate the ability of ODB to rapidly and easily test new anomaly detection algorithms.

However, in order to perform this test, a dataset with known anomalies was needed. While several popular solutions exist, none of them were suitable for this test. The most popular, the DARPA dataset, has many known issues, and is very old. Most others were not available or had their own issues. To address this issue, a new dataset was created, partnering with MERLIN to create a dataset that involves as much real-world traffic as possible, and relying on injected attacks on known hosts to provide the anomalies.

In testing, ODB performed very well in both memory efficiency and throughput, in both cases exceeding the requirements laid out for it. It is able to process 10 million packets in 25s on the testing hardware, or about 400 thousand packets per second, almost double the requirement for a gigabit network, all while consuming only about 2% memory overhead.

ODB helped demonstrate that the algorithm designed by Lakhina et al. is not suitable for use on smaller networks; using packet-level data. While there is still opportunity for improvement and future work on ODB, other researchers will find it useful for testing their own anomaly detection algorithms.

Appendix A

Network Attack Schedule

A.1 Host A

Start time	Command	Duration (seconds)
Dec 09 22:34:24	/usr/bin/nmap -PS -sS -p 1-1024 216.73.66.48/28	1329.597
Dec 10 07:13:57	/usr/bin/nmap -PS -sS -p 1-1024 142.13.0.0/16	5327.916
Dec 10 12:23:20	/usr/bin/python ssh-brute/brutessh/brutessh.py -h 216.73.66.51 -u root -d ssh-brute/passlist.txt	980.140
Dec 10 18:51:07	/usr/bin/nmap -PS 142.13.0.0/16	5282.882
Dec 11 07:53:35	/usr/bin/nmap -PS -sS -p 1-1024 142.13.0.0/16	3048.166
Dec 11 16:02:33	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	1767.875

Dec 11 22:54:57	/usr/bin/nmap -PS -sS -p 1-1024 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	1512.697
Dec 12 01:33:55	/usr/sbin/mz eth0 -A rand -B 216.73.66.50 -c 100000 -t tcp dp=80,flags=syn,sp=63431	819.533
Dec 12 02:49:31	/usr/bin/nmap -PS 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	851.100
Dec 12 11:14:00	/usr/bin/nmap -PS 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	716.537
Dec 12 12:53:17	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	1120.030
Dec 12 21:29:05	/usr/bin/nmap -PS 142.13.0.0/16	4803.881
Dec 13 02:42:50	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	10787.203

Dec 13 16:36:42	/usr/sbin/mz eth0 -A rand -B 216.73.66.51 -c 100000 -t tcp dp=80,flags=syn,sp=53857	684.739
Dec 14 01:13:09	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	1796.462
Dec 14 12:30:21	/usr/bin/nmap -PN -sS -p 1-1024 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	833.682
Dec 14 19:26:47	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	669.611
Dec 14 22:22:09	/usr/bin/nmap -PS 216.73.66.48/28	2151.366
Dec 15 07:26:25	/usr/bin/python pyloris-3.2/httploris.py 216.73.66.50 -w 0.1 -W 0.1	3545.194
Dec 15 13:19:21	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	5593.238
Dec 15 13:46:39	/usr/bin/nmap -PS 216.73.66.48/28	2708.623
Dec 15 22:44:52	/usr/bin/nmap -PS 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	7820.550

Table A.1: Attacks from Host A

A.2 Host B

Start time	Command	Duration (seconds)
Dec 09 00:21:14	/usr/bin/nmap -PS -sS -p 1-1024 142.13.0.0/16	4170.704
Dec 09 03:36:37	/usr/bin/python pyloris-3.2/httploris.py 216.73.66.50 -w 0.1 -W 0.1	3797.274
Dec 09 04:16:30	/usr/sbin/mz eth0 -A rand -B 216.73.66.51 -c 100000 -t tcp dp=22,flags=syn,sp=2190	5684.059
Dec 09 04:45:45	/usr/bin/nmap -PN -sS -p 1-1024 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	652.992
Dec 09 05:08:21	/usr/bin/python ssh-brute/brutessh/brutessh.py -h 216.73.66.51 -u root -d ssh-brute/passlist.txt	3634.524
Dec 09 15:46:07	/usr/sbin/mz eth0 -A rand -B 216.73.66.50 -c 100000 -t tcp dp=22,flags=syn,sp=55036	726.915
Dec 09 20:18:35	/usr/bin/nmap -PS 142.13.0.0/16	807.328
Dec 10 12:45:19	/usr/bin/nmap -PS 216.73.66.48/28	2314.561

Dec 12 03:30:53	/usr/bin/nmap -PN -sS -p 22 -sV 216.73.66.48/28	654.883
Dec 12 05:12:49	/usr/bin/nmap -PN -sS -p 1-1024 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	1173.956
Dec 12 18:57:27	/usr/sbin/mz eth0 -A rand -B 216.73.66.50 -c 100000 -t tcp dp=80,flags=syn,sp=59761	5462.673
Dec 13 13:28:05	/usr/bin/nmap -PS 216.73.66.48/28	1649.023
Dec 13 23:28:50	/usr/bin/nmap -PS 216.73.66.48/28	3315.546
Dec 14 08:03:48	/usr/bin/python pyloris-3.2/httploris.py 216.73.66.50 -w 0.1 -W 0.1	4236.416
Dec 14 08:18:52	/usr/bin/nmap -PN -sS -p 1-1024 216.73.66.48/28	1631.494
Dec 14 20:42:48	/usr/bin/nmap -PN -sS -p 1-1024 216.73.66.48/28	798.834
Dec 15 00:28:17	/usr/bin/nmap -PS 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	1898.930

Table A.2: Attacks from Host B

A.3 Host C

Start time	Command	Duration (seconds)
Dec 10 13:14:40	/usr/bin/nmap -PS 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	9730.082
Dec 10 14:25:56	/usr/bin/nmap -PN -sS -p 1-1024 142.13.16.0/20 142.13.48.0/23 142.13.50.0/24 142.13.80.0/20 142.13.141.0/24 142.13.147.0/24 142.13.148.0/22 142.13.156.0/23	836.673
Dec 10 14:47:09	/usr/bin/nmap -PS 216.73.66.48/28	911.502
Dec 10 15:09:30	/usr/bin/python ssh-brute/brutessh/brutessh.py -h 216.73.66.50 -u root -d ssh-brute/passlist.txt	1580.785
Dec 11 00:41:01	/usr/bin/python pyloris-3.2/httploris.py 216.73.66.51 -w 0.1 -W 0.1	3213.516
Dec 11 02:57:58	/usr/bin/python pyloris-3.2/httploris.py 216.73.66.51 -w 0.1 -W 0.1	5273.939

Dec 12 00:42:18	/usr/bin/python ssh-brute/brutessh/brutessh.py -h 216.73.66.50 -u root -d ssh-brute/passlist.txt	4585.136
Dec 13 11:54:21	/usr/bin/nmap -PS -sS -p 1-1024 216.73.66.48/28	1827.026
Dec 14 20:24:32	/usr/bin/nmap -PN -sS -p 1-1024 216.73.66.48/28	2510.102
Dec 15 04:15:25	/usr/bin/python ssh-brute/brutessh/brutessh.py -h 216.73.66.50 -u root -d ssh-brute/passlist.txt	1006.071
Dec 15 14:48:21	/usr/sbin/mz eth0 -A rand -B 216.73.66.51 -c 100000 -t tcp dp=80,flags=syn,sp=44250	8160.078

Table A.3: Attacks from Host C

References

[1] Lawrence G. Roberts. Beyond Moore's Law: Internet Growth Trends. *IEEE Computer*, 33(1):117–119, 2000. URL http://dblp.uni-trier.de/db/journals/computer/computer33.html#Roberts00.

[2] K. Thompson, G.J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *Network, IEEE*, 11(6):10–23, 1997. ISSN 0890-8044. doi: 10.1109/65.642356.

[3] Kerry Coffman and Andrew M. Odlyzko. Internet growth: is there a "Moore's law" for data traffic? *Handbook of massive data sets*, pages 47–93, 2002. URL http://www.bibsonomy.org/bibtex/22652f29fa14667a6148b43846e5219cb/bsmyth.

[4] Sensage. A Practical Guide to Next Generation SIEM. http://www.sensage.com/sites/default/files/sens_gd_next-gen_siem_03ol.pdf.

[5] Infosecurity. Hackers are winning the cat-and-mouse game against anti-virus programmers. http://www.infosecurity-magazine.com/view/11690/hackers-are-winning-the-catandmouse-game-against-antivirus-programmers/.

[6] Michael Himbeault and Paul Card. Personal correspondance.

[7] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining Anomalies Using Traffic Feature Distributions. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *SIGCOMM*, pages 217–228. ACM, 2005. ISBN 1-59593-009-4. URL http://dblp.uni-trier.de/db/conf/sigcomm/sigcomm2005.html#LakhinaCD05.

[8] Anukool Lakhina, Mark Crovella, and Christophe Diot. Characterization of Network-wide Anomalies in Traffic Flows. In Alfio Lombardo and James F. Kurose, editors, *Internet Measurement Conference*, pages 201–206. ACM, 2004. ISBN 1-58113-821-0. URL http://dblp.uni-trier.de/db/conf/imc/imc2004.html#LakhinaCD04.

[9] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing Network-wide Traffic Anomalies. In Raj Yavatkar, Ellen W. Zegura, and Jennifer Rexford, editors, *SIGCOMM*, pages 219–230. ACM, 2004. ISBN 1-58113-862-8. URL http://dblp.uni-trier.de/db/conf/sigcomm/sigcomm2004.html#LakhinaCD04.

[10] Anukool Lakhina, Konstantina Papagiannaki, Mark Crovella, Christophe Diot, Eric D. Kolaczyk, and Nina Taft. Structural Analysis of Network Traffic Flows. In Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors, *SIGMETRICS*, pages 61–72. ACM, 2004. ISBN 1-58113-873-3. URL http://dblp.uni-trier.de/db/conf/sigmetrics/sigmetrics2004.html#LakhinaPCDKT04.

[11] Anthony M Freed. Why SIEM Alone is Not Enough. http://www.tripwire.com/state-of-security/incident-detection/why-siem-alone-is-not-enough/.

[12] misnomer. SIEM – The Good, The Bad and The Ugly. http://infosecnirvana.com/siem-technology-a-critical-analysis/.

[13] Thor Olavsrud. Targeted Attacks Increased, Became More Diverse in 2011. www.cio.com/article/705334/Targeted_Attacks_Increased_Became_More_Diverse_in_2011.

[14] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Martin May, and Anukool Lakhina. Impact of packet sampling on anomaly detection metrics. In Jussara M. Almeida, Virgílio A. F. Almeida, and Paul Barford, editors, *Internet Measurement Conference*, pages 159–164. ACM, 2006. ISBN 1-59593-561-4. URL http://dblp.uni-trier.de/db/conf/imc/imc2006.html#BrauckhoffTWML06.

[15] Peter Zaitsev. High Rate insertion with MySQL and Innodb. http://www.mysqlperformanceblog.com/2011/01/07/high-rate-insertion-with-mysql-and-innodb/, 2011.

[16] Oracle. MySQL 5.1 Reference Manual. http://dev.mysql.com/doc/refman/5.1/en/load-data.html.

[17] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2010.

[18] Tokyo Cabinet: a Modern Implementation of DBM. http://fallabs.com/tokyocabinet/, 2010.

[19] Adrian Cockcroft and Denis Sheahan. The Netflix Tech Blog: Benchmarking Cassandra Scalability on AWS - Over a million writes per second . http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html, November 2011.

[20] Wolfgang Hummel. Large Scale High Performance OpenLDAP. ldapcon.org/downloads/hummel-slides.pdf, 2011.

[21] LDAP for Rocket Scientists. http://www.zytrax.com/books/ldap/ch2.

[22] Snort. www.snort.org.

[23] Doug Dineley and High Mobley. The greatest open source software of all time. *InfoWorld*, August 2009.

[24] HP TippingPoint Next Generation Intrusion Prevention System (NGIPS). http://www8.hp.com/us/en/software-solutions/software.html?compURI=1343617.

[25] Wireshark. http://www.wireshark.org.

[26] David Chappelle et al. gperftools. http://code.google.com/p/gperftools/.

[27] Ted Nyman. TCMalloc and MySQL. https://github.com/blog/1422-tcmalloc-and-mysql.

[28] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. URL http://dblp.uni-trier.de/db/journals/csur/csur15.html#HarderR83.

[29] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *VLDB*, pages 144–154. IEEE Computer Society, 1981. URL http://dblp.uni-trier.de/db/conf/vldb/vldb81.html#Gray81.

[30] Matthew Mahoney and Philip Chan. An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In Giovanni Vigna, Christopher Kruegel, and Erland Jonsson, editors, *Recent Advances in Intrusion*

Detection, volume 2820 of *Lecture Notes in Computer Science*, pages 220–237. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40878-9. URL http://dx.doi.org/10.1007/978-3-540-45248-5_13. 10.1007/978-3-540-45248-5_13.

[31] John McHugh. Testing Intrusion Detection Systems: a Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. *ACM Trans. Inf. Syst. Secur.*, 3:262–294, November 2000. ISSN 1094-9224. doi: 10.1145/382912.382923. URL http://doi.acm.org/10.1145/382912.382923.

[32] Herbert Haas. Mausezahn. http://www.perihel.at/sec/mz.

[33] Wong Onn Chee and Tom Brennen. Layer 7 DDoS. https://www.owasp.org/images/4/43/Layer_7_DDOS.pdf, November 2010.

[34] John Kinsella, Hugo Gonzales, et al. Slowloris. http://ckers.org/slowloris/.

[35] Christopher Gilbert et al. PyLoris. http://motoma.io/pyloris/.

[36] Christian Martorella, Xavier Mendez, et al. edgessh. http://code.google.com/p/edgessh/.

[37] Haining Wang, Danlu Zhang, and Kang G. Shin. Detecting SYN Flooding Attacks. In *INFOCOM*, 2002. URL http://dblp.uni-trier.de/db/conf/infocom/infocom2002.html#WangZS02;http://www.ieee-infocom.org/2002/papers/800.pdf;http://www.bibsonomy.org/bibtex/2ae50f58ec33e6c3126cffa7e45e48219/dblp.

[38] A. Nucci and S. Bannerman. Controlled Chaos [Internet Security]. *Spectrum, IEEE*, 44(12):42–48, 2007. ISSN 0018-9235. doi: 10.1109/MSPEC.2007.4390022.

[39] R. Linsker. Self-organization in a perceptual network. *Computer*, 21(3):105–117, 1988. ISSN 0018-9162. doi: 10.1109/2.36.

[40] Michael Niedermayer. `http://guru.multimedia.cx/index.php?s=pca`.