TUPLE FILTERING IN SILK USING CUCKOO HASHES

by

Aaron Webb

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2010

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled “TUPLE FILTERING IN SILK USING CUCKOO HASHES” by Aaron Webb in partial fulfillment of the requirements for the degree of Master of Computer Science.

Dated: August 20, 2010

Supervisor: _______________________________

Readers: _______________________________

____

To Catherine,

You have more patience than you give yourself credit for.

Table of Contents

List of Tables

List of Figures

Abstract

SiLK Tools is a suite of network flow tools that network analysts use to detect intrusions, viruses, worms, and botnets, and to analyze network performance. One tool in SiLK is tuple filtering, where flows are filtered based on inclusion in a "multi-key" set (MKset) whose unique members are composite keys whose values are from multiple fields in a SiLK flow record. We propose and evaluate a more efficient method of implementing MKset filtering that uses cuckoo hashes, which underlie McHugh et al.'s cuckoo bag (cubag) suite of MKset SiLK tools. Our solution improves execution time for filtering with an MKset of size k by a factor of $O(logk)$, and decreases memory footprints for MKset filtering by 50%. The solution also saves 90% of disk space for MKset file storage, and adds functionality for transformations such as subnet masking on flow records during MKset filtering.

Acknowledgements

Thanks go to my supervisor, Dr. Mike McAllister, for sharing his steady encouragement, clear thinking, and wealth of experience. Also to Randy Currie and Solution-Inc, Ltd., for their tireless support during my time there and beyond. To Dr. John McHugh goes my gratitude for great ideas and a codebase to work with. Finally, thanks to Ron McLeod for immersing me in the world of network flow and SiLK analysis.

Chapter 1

Introduction

Analysis of network traffic is an important part of network maintenance, security, and infrastructure planning. Concise measurements of the types and volumes of traffic passing through a network provide network managers with quantitative data that help to organize network structure, defend against attacks, and plan upgrades to network hardware and software.

As traffic volumes on local networks and the Internet have increased, and with the rise of encrypted data transfer, analysis of the payload contents (e.g., deep packet inspection) of all traffic is quickly becoming unmanageable and intractable for many analysis purposes and some higher-level analysis is desirable. One solution to gaining a broader view of network activity with a manageable amount of data is to collect "network flow" data, which records routing information about conversations between applications on host computers without retaining the actual contents of every packet passing through the network.

Network flow analysis is a technique for examining traffic patterns and behaviour on networks. By focusing on routing information in packet headers instead of the packet payload, we can store complete network traffic information using relatively small amounts of disk space. Network flow further reduces analysis data by aggregating network traffic into individual conversations, or "flows", between host machines, based on IP addresses, ports, and protocol. In addition to these identifying attributes, network flow records contain information such as the number of packets and bytes transmitted, start and end times, and a number of other aggregate information [5]. By focusing on the "who", "where", and "when" of network traffic, and moving the "what" from individual packets to the broader context of flows, we can gain insight into the "why" of various network issues and behaviours.

Network flow analysis is used by people such as network managers and security analysts to monitor [34], plan, and maintain [25] IP networks. Analysis ranges in scope from profiling hosts [16] to monitoring overall network statistics [35]. They employ varying data mining techniques to solve problems such as detecting specific host application activity (e.g., peer-to-peer software, botnet robots, FTP and Web servers) [10], profiling different types of traffic (e.g., TCP vs UDP), ensuring satisfaction of service-level agreements or SLAs [31] (e.g., data throughput guarantees), and targeting network hardware for infrastructure changes (e.g., adding new routers or switches in high-traffic sections of a network).

Network flow is generally used as a forensics tool, with flow records stored on disk for months' and even years' worth of network traffic. As such, the quantities of flow data that analysts work with can be very large, and a set of efficient tools is required to search for and analyze flow records. Developed by the CERT Network Situational Awareness team (CERT NetSA), the SiLK (System for Internet-Level Knowledge) network flow collection and analysis suite [3] is one such set of efficient tools; others include JFlow [18], SFlow [24], and Argus [26].

The SiLK suite includes a tool to filter network flow records based upon inclusion in a set of tuples, which are comprised of arbitrary subsets of the fields found in a flow record. Filtering can assist in tasks as simple as extracting all flows from a given subnet, to complex analysis such as detecting network activity from specific types of host operating systems or software. For instance, one might build a set of tuples containing the destination IPs (DIPs), destination ports (DPORTs), and protocols used by known FTP servers on a network, and use this set to remove FTP traffic from flow data before searching the data for peer-to-peer (P2P) activity. For clarity, we shall denote the general notion of a set of tuples, each containing multiple SiLK flow fields, as a "multi-key set", or "MKset" for short, and filtering SiLK flow data with an MKset shall be called "MKset filtering". To remain consistent with the current SiLK terminology, we will call the SiLK tools' version of a MKset a "tuple", and SiLK's current method of MKset filtering "tuple filtering".

As suggested by Kumar [12], the MKset filtering functionality in SiLK currently has limitations in functionality and efficiency that warrant investigation into improvements in file storage, MKset creation, filtering options, internal algorithms, and data structures. Building on work done by McHugh et al. [15], we present an improved MKset filtering mechanism in SiLK using a search structure called a cuckoo hash [23].

McHugh et al. call their MKsets "cusets" (short for "cuckoo sets"); we follow this convention, and call MKset filtering with cusets "cuset filtering".

The goal in this work is to provide an improved means for network flow analysts to perform MKset filters on SiLK flow data. The key desired improvements over SiLK's current MKset filtering capabilities are three-fold:

1. faster execution times
2. decreased memory usage
3. added functionality for manipulation of set keys

These improvements are desirable both when building MKsets and using them to filter SiLK data.

As additional considerations, we wish to work within the following constraints to help make our work as accessible as possible to users of the SiLK toolset:

- Maintain or improve the numbers of commands and non-SiLK scripting required to generate, store, and filter MKsets
- Assure that any new tools or methods fit within the standard flow of the SiLK tools
- Consider that we may want to modify or merge MKsets to allow greater filter flexibility, similarly to SiLK's current bag and set tools

The main contribution of this dissertation is in providing faster and more flexible filtering to network analysts, which allows the analysts to explore their network traffic data in a more iterative and flexible manner. Specifically, by altering the SiLK tools code to allow MKset filtering with cuckoo bags using cuckoo hashing, we provide network analysts with an improved tool to replace SiLK's tuple filtering. Cubag filtering is faster and uses less memory than tuple filtering, and provides added functionality for manipulation of MKset keys during the creation and filtering processes.

SiLK's tuple filtering mechanism does not allow for filtering of flows containing IPv6 address fields, and cubag filtering expands MKset filtering in SiLK to fully support filtering IPv6 flows and MKsets containing IPv6 keys. In addition, while the library used to build and search MKsets uses SiLK's Netflow-specific data structures, replacement of these data structures with more generalized structures would provide a code base on which other, non-flow-specific MKset filtering could be implemented. This would entail some modifications to the actual hash library functions as well, but the algorithms and framework should generalize well to filtering of any other types of data such as MAC addresses or even non-networking-specific data.

The organization of the thesis is as follows: network flow using SiLK tools, including current methods for MKset filtering, are discussed in Chapter 2. Chapter 3 outlines the steps taken to design, implement, and validate an improved method for MKset filtering in SiLK. Chapter 4 contains our analyses of data structures and the SiLK application framework in which our solution was developed. Chapter 5 presents the results of our experimental validation, and Chapter 6 provides concluding remarks and future directions for this work.

Chapter 2

Background

To aid in our explanation and analysis of MKset filtering in SiLK, we present an overview of network flow and the SiLK tools suite. The notion of MKset filtering is explained, along with some information on cuckoo hashes and a related structure, the Bloom Filter. Finally, we present previous works on operations similar to MKset filtering, and provide motivation for improved MKset filtering by describing some tools with which it could be used.

2.1 Network Flow

Network flow is a technique created by Cisco Systems [4] to gather and analyze summarized information about traffic on IP networks. Using routing information from packet headers, network flow tracks and stores information on conversations between network hosts, or "flows". A "network flow" is defined as a unidirectional sequence of packets that share the following five attributes:

- Source IP address
- Destination IP address
- Source port
- Destination port
- IP protocol

Flows are uniquely identified on a given network by these five attributes, and the flow start and end times. There are several variations on network flow collection and analysis techniques, including JFlow [18], SFlow [24], Argus [26], and NetFlow [4] (and NetFlow's industry standard offspring, IPFIX [5]). Some of these, such as Argus and IPFIX, include the ability to mix deep packet inspection with flow analysis; our work concerns only flow analysis. While implementations of these protocols abound, our work focuses on the System for Internet-Level Knowledge, or SiLK, collection and analysis suite [3].

2.2 SiLK

Developed and maintained by the CERT Network Situational Awareness (NetSA) team and researchers from Carnegie Mellon University, the SiLK suite [3] offers a comprehensive set of tools for collection, storage, and analysis of network flow data.

The SiLK suite uses a network flow format that is based on Cisco NetFlow [2], which was created by Cisco Systems to run on its routers; however, as NetFlow is supported by other platforms and SiLK can collect and convert network flow in other formats, we use the more general terms "network flow" (or simply "flow", for brevity) when referencing SiLK flow records.

A SiLK flow record contains a variety of information about the flow, as itemized in Table 2.1 [3]. A SiLK data store consists of SiLK flow records stored in a time and date-indexed file and directory structure. For example, SiLK flow records with an "sTime" value of August 16, 2009 between 1pm and 2pm would be stored in the SiLK file /path/to/data/store/2009/08/16/20100816.13.

The SiLK suite contains a number of tools to display, filter, and analyze flow records. The flow of analysis in SiLK normally consists of using a tool called rwfilter to select SiLK flow records from a data store, and either piping the selected records to other SiLK tools for analysis or storing them in a file for subsequent analyses. For example, the following command generates a list of the ten most commonly-occurring source IP addresses in all TCP flows on August 16, 2009:

```
rwfilter --start-date=2009/08/16 --end-date=2009/08/16
   --protocol=6 --pass=stdout | rwstats --fields=sIP
   --count=10
```

In this example, the --start-date and --end-date parameters tell rwfilter which SiLK records to read from the data store, and --protocol=6 specifies that we want flows for TCP traffic only. --pass=stdout causes rwfilter to write the flows matching the date and protocol parameters to the standard output stream, which is then piped to the rwstats tool. Finally, the --fields=sIP and --count=10 cause rwstats to count

Field Name	Description
sIP	Source IP address for flow record
dIP	Destination IP address for flow record
sPort	Source port (or ICMP type) for flow record(or 0)
dPort	Destination port (or ICMP code) for flow record(or 0)
protocol	Protocol number for flow record
packets	Number of packets in flow
bytes	Number of bytes in flow
flags	Logical or of TCP flag fields of flow (or blank)
sTime	Start date and time of flow (in seconds)
dur	Duration of flow (in seconds)
eTime	End date and time of flow (in seconds)
sensor	Sensor that collected flow
nhIP	Next hop IP address (currently used only for annotations)
stype	Source group of IP addresses (pmap required)
dtype	Destination group of IP addresses (pmap required)
scc	Source Country Code (pmap required)
dcc	Destination Country Code (pmap required)
class	Class of sensor that collected flow
type	Type of flow for this sensor class
sTime+msec	Start date and time of flow (in milliseconds)
eTime+msec	End date and time of flow (in milliseconds)
dur+msec	Duration of flow (in milliseconds)
icmpTypeCode	ICMP type and code values
InitialFlags	TCP flags in Initial Packet
SessionFlags	TCP flags in remaining Packets
attributes	Constants for termination conditions
application	Standard port for application that produced traffic

Table 2.1: SiLK Flow Fields [3]

flows for each unique source IP address (SIP) in the selected data, and print a list of the top ten SIPs and their associated flow counts. A selection of commonly-used SiLK commands is shown in Table 2.2.

Many of the analysis tasks performed by SiLK analysts involve creating and maintaining sets of specific network parameters or behaviours, as defined by a set of values for a subset of SiLK flow fields. For instance, one might wish to maintain a set of tuples with format (IP address, protocol, port) for hosts running a specific client or server applications such as FTP, P2P, or SSH. As these sets contain keys consisting of multiple SiLK fields, we shall call such a set a “multi-key set”, or MKset. We define a multi-key set as follows:

Command	Description
rwfilter	Select SiLK flow records from a file or data store, based on specified values for any of the SiLK flow fields
rwcut	Print selected fields of binary SiLK flow records in ASCII text format
rwsort	Sort SiLK Flow records on one or more fields
rwset	Builds a set of unique source, destination, or next-hop IP addresses from SiLK flow records
rwbag	Builds a bag of (unique key, counter) entries from SiLK flow records.
rwtotal	Count how much traffic matched specific keys
rwcount	Print traffic summary across time
rwstats	Print interval counts or top-N/bottom-N lists
rwuniq	Bin SiLK flow records by a key and print the bins' volume

Table 2.2: Commonly-Used SiLK Tools

A "multi-key" is an entity containing values in the key from one or more SiLK flow fields. A "multi-key set", or "MKset", is a set containing one or more unique multi-key elements.

While viewing and analyzing MKsets on their own has value, the ability to filter flow records using stored MKsets gives the analyst the power to both remove sets of known "normal" traffic when searching for anomalous data, and to use a given set as a fingerprint to search for a given network behaviour. This work focuses on filtering SiLK flows using MKsets, which we shall call "MKset filtering".

As an example, consider the FTP traffic filter mentioned in the introduction. The technique comes from the SiLK tooltips site [3], and is used to identify FTP traffic in SiLK flows either to analyze the FTP traffic itself or remove it from SiLK flows to speed up an analysis in which FTP traffic is not important to the analyst. rwfilter is used to find flows containing the "short" command, which initiates every FTP transfer, by searching for flows with an sPort value of 21 (the standard port for initializing FTP transfers), protocol of 6 (TCP), and two or more packets. These flows are used to build an MKset containing the SIP and DIP of all hosts participating in FTP transfers. Finally, an MKset filter is performed to find all outbound FTP traffic by searching for TCP flows from FTP source ports (20,21, or 1024+) to any destination port greater than 1024, and with the (SIP, DIP) pair matching the MKset. A similar process is used to find incoming FTP traffic. Using rwfilter's −−pass or −−fail arguments, all FTP flows in the selected SiLK flows can either be all the FTP

```
            sIP|            dIP| pro
    10.0.45.214|    189.7.83.12|  17
    10.0.45.111|    64.4.35.253|  17
     10.0.45.48|       10.0.0.1|   6
   74.86.125.37|    10.0.45.164|   1
     10.0.45.18|     64.4.36.59|   6
     10.0.45.69|   98.137.80.32|   6
 64.233.161.164|    10.0.45.109|   6
 78.140.149.103|     10.0.45.77|   6
    10.0.45.137|  38.108.99.201|   6
     10.0.45.73| 212.58.226.142|   6
```

Figure 2.1: Tuple file with 10 keys

flows that were found, or everything but the FTP flows. The resulting flows can then be stored for further analysis.

The current SiLK method to perform an MKset filter is called “tuple filtering”, and is performed by specifying a “tuple file” to rwfilter with the −−tuple−file option. A SiLK tuple file is a text file consisting of a header row that specifies the SiLK fields used in the file (as per Table 2.1), followed by one tuple record (SiLK’s version of a multi-key) per line. The standard method to create a tuple file in SiLK uses the rwuniq tool, which uses a hash table to build bins of unique SiLK field values in a set of SiLK flow records, and prints the bins and the number of flows in each bin in ASCII text format. The output from rwuniq must then be processed by an external utility such as the Unix “cut” command to generate a SiLK tuple file. This process is detailed in Chapter 5. Figure 2.1 shows an example of a tuple file.

When executing a tuple filter, rwfilter reads a tuple file into a red-black tree [6], and searches this tree to determine whether each input flow is present in the tuple file. Records found in the tuple file can then be sent to either of rwfilter’s “pass” or “fail” output streams, to be saved on disk or piped into further processing commands.

One drawback of SiLK’s tuple filtering is that creation of a tuple file must be performed by the analyst, either by hand or by formatting the output of the other SiLK text dump tools using external utilities. In addition, using a text-based file format results in slower read and write times than a binary file in SiLK’s native format would allow.

The main aspect of SiLK tuple filtering that we have identified for improvement is execution time. A red-black tree can perform insert and search operations in $O(\log n)$ time and allows for fast deletion time (also $O(\log n)$) [6]. We propose to replace the red-black trees with a data structure with faster performance for insertion and search, and allowing for slower deletion times.

A cuckoo hash, as detailed in Chapter 4, is named after the parasitic Cuckoo bird, whose newly-hatched young will push eggs out of a host bird's nest. The cuckoo hash addresses hash table collisions by employing multiple hash functions. During an insertion, if a key k is hashed to an occupied location, the key residing there is "pushed out" and re-hashed using a different hash function. The insert/re-hash process continues until either an empty spot is found for an evicted key, or K evictions occur where K is the number of keys in the hash table, and the hash table must be rebuilt. With constant ($O(1)$) amortized insert and search times [23], the cuckoo hash is one logical choice to improve upon SiLK's tuple filtering functionality.

Additions to the SiLK suite by McHugh et al. [15] have provided a means of building, manipulating, and viewing multi-key sets directly from a SiLK data store, with disk storage in a native binary format, using cuckoo hashing. These MKsets are referred to as "cuckoo sets", or cusets for short. McHugh's work extended to multi-key bags, and hence the new tool suite refers mainly to "cubags" in its tool names; our work focuses on the cuset features of the cubag additions.

A cuset is similar in concept to a SiLK tuple file, as previously described. Cusets are built using McHugh's cubag tool (see Appendix A), which generates cuset files based on aggregated SiLK data from user-specified SiLK flow fields. For example, the following command would generate a cuset file called sip_dip_protocol.cu, similar to the tuple file from Figure 2.1, from a SiLK input file called example.rwf:

```
cubag --set-file=sip_dip_protocol.cu:sIP,dIP,protocol example.rwf
```

In addition to the tuple-like MKset creation ability, the cubag tool allows the user to apply modifications to SiLK data during cuset creation, such as bit masking and bit shifting. For instance, we could modify the above example to create a cuset file that stores the class-B subnet of each sIP instead of the entire IP address, by masking the lower 16 bits as follows:

```
cubag
    --set-file=sip_dip_protocol.cu:sIP(&,255.255.0.0),dIP,protocol
    example.rwf
```

Extending McHugh's work to add direct integration with rwfilter for performing an MKset filter would provide an alternative to SiLK's tuple filtering. MKset filtering with cusets provides a potential speed improvement of cuckoo hashing over red-black trees, and faster and more efficient data storage and retrieval of the binary cubag file over an ASCII text tuple file. Also, using a specific SiLK tool for the generation of the multi-key set requires no user manipulation of SiLK output, and has the added feature of key modification during the cubag building and filtering processes. Through these improvements, we will show that the addition of a cubag search to SiLK's rwfilter tool provides an improved means for flow analysts to perform multi-key set filters on SiLK data.

McHughs work investigated using Bloom filters [1] to create MKsets for fast execution of MKset searches. The Bloom filter is a probabilistic data structure that uses an array of bits to store information about set membership. An element x is added to an m-bit filter by computing $h_i(x)$ for k hash functions $(h_1, h_2, ..., h_k)$ that map x to locations in the bit array, and setting those bits to 1. To test for set membership of an element, the hash functions are computed for the element, and if any bit array location computed by the hash functions is not set, then the element is not in the set. If all of its bits are set in the array, then either the element exists in the set, or its bit array locations happen to have been set by insertions of other elements into the set (a false positive). Deletions are not allowed, which ensures that no false negatives occur in membership testing.

Increasing the ratio m/n of the size of the bit array m to the number of elements n, and the number of hash functions k, reduces the rate of false positives at the cost of increased memory usage. For instance, with five hash functions and a bit array ten times larger than the number of elements to filter (i.e., $k = 5$ and $m/n = 10$), we get a false positive rate of 0.9% [8]. However, to match the functionality of SiLK's tuple filtering, whose output is the exact subset of input flows matching keys in an MKset, even a single false positive during filtering is unacceptable.

2.3 Related Work

Operations similar to MKset filtering are used for tasks such as network visualization, classification of peer-to-peer and other application-specific traffic, and detection of intrusion and anomalous network behaviour. We present previous work describing some similar techniques to MKset filtering, and provide motivation for improved MKset filtering of NetFlow data by describing systems which could benefit from our cuckoo hashing solution.

Autofocus [7] is a traffic analysis and visualization tool that uses clustering of NetFlow data to display visual characterizations of network behaviour. Autofocus creates “multidimensional clusters” of network traffic using hierarchical, multidimensional graphs - i.e., trees. It uses these trees to create visual reports of network behaviour and compute the “unexpectedness”, or likelihood, of a particular subset being present in a set of NetFlow data. Lakhina et al. [13] use a measure of entropy to cluster NetFlow data, and the clusters it creates, which are comparable to MKsets, are used for automatic classification in an anomaly detection system. However, as mentioned in Section 2.2, we require MKset filtering to create exact subsets of flows matching MKset keys, so these probabilistic approaches are unsuitable for our use.

Nickless [21] uses a relational database to store NetFlow data and allow SQL queries for analysis tasks. MKset queries could be designed in SQL using sub-select statements, but adapting SiLK to use an RDBMS violates our constraint of maintaining the current flow of SiLK analyses and sacrifices execution speed due to RDBMS overhead.

Possible Uses for Cubag Filtering

FlowRank [33] uses Google’s PageRank [22] algorithm to rank NetFlow records from the perspective of bandwidth consumption, threat likelihood, and revenue generation. Cubag filtering could be used as a post-processing mechanism for FlowRank to help find flows that match a set of FlowRank-ranked flows representing different characterizations of the bandwidth, threat, and revenue rankings.

Nagaraj [19] uses hash functions to perform computation of “aggregates” based on “group-by” attributes, which is similar to creation of tuple or cubag-based MKsets.

Aggregates are filtered using a tree structure, which might be improved using our cuckoo hashing technique.

MIND [14] and SkipIndex [36] both use prefix hash trees [27] (PHT) and distributed hash tables [29] (DHT) to perform multidimensional range searches on multiple NetFlow data stores. MIND is focused on anomaly detection, and SkipIndex is used to create peer-to-peer indexing services. Cubag filtering could be used with both MIND and SkipIndex to perform base-level searches of NetFlow data after the PHT and DHT structures have been used to narrow down searches to a particular NetFlow data store.

Snort [28] uses aggregated network information from packet captures to perform network intrusion prevention and detection. Snort uses a two-dimensional linked list to search for packets matching signatures of known attack methods. Using cubags as intrusion signatures, a NetFlow-based system similar to Snort is conceivable, which could allow broader analysis by leveraging the reduced data volumes of NetFlow as compared to packet captures.

Chapter 3

Approach

The main goal in this work is to provide an improved means for network flow analysts to perform MKset filters on SiLK flow data. To that end, the following steps were taken.

1. **Identify the best context within which to deploy a solution in SiLK.**

   In development by the CERT NetSA team since 2003, SiLK has a fairly mature code base and a well-established flow for network data analysis. We wish to maintain that flow with any new tools we create, and when possible, leverage any existing libraries and methods for doing menial tasks such as option processing, file I/O, and operations on SiLK data.

2. **Identify the profile of the data structure usage that will fit into the chosen context, and the range of data sizes that the design should consider.**

   Given the overall framework of our MKset filtering method, choice of a data structure to represent a MKset is guided by details about how and when insertion, deletion, sorting, and search operations are performed. Knowledge of the size and composition (such as heterogeneity) of the network flow data to be analyzed will help narrow down the type of structure to employ.

   Most important to our results are operations performed while filtering an MKset against a SiLK data stream. However, ideally, the chosen structure should also be compatible with an MKset data representation that allows creation of new and/or usage of existing fast and flexible tools for building, modification, and merging of MKsets to be used for filters.

3. **Identify properties of the current data structure used by SiLK's MKset filtering.**

Theoretical analysis of the current implementation for SiLK's tuple file filtering capability, compared with some alternate methods, gives us an informed choice for an improved solution.

4. **Choose a data structure to improve filtering performance.** Using our statement of desired performance improvements in our chosen framework, the makeup of the data to be analyzed, and the nature of a potentially better data structure, we can select a structure based on the expected asymptotic performance.

5. **Implement the chosen solution as a prototype in SiLK.**

   Integration of our chosen design decisions into the SiLK Tools code base provides the opportunity to compare actual performance against that of the current method and analyze any new functionality we have provided.

6. **Validate the implementation experimentally against the theoretical expectations.**

   Watching primarily for execution time and memory footprint as success factors for the basic implementation, we can determine whether our predicted performance improvements worked in practice, and analyze why our solution did or did not perform as expected. We also compare the performance with different modifiers to assess the cost of the added functionality to MKset filters.

Chapter 4

MKset Filtering Solution

When designing an improved MKset filtering solution in SiLK, implications of the following aspects were considered:

- Code framework in which to implement the solution
- Use characteristics of MKset filtering
- Candidate data structures to replace SiLK's red-black trees
- Details of previous work by McHugh et al. [15] to implement cuckoo hashing in SiLK
- Ability to export the solution to other applications

4.1 Framework

rwfilter is the cornerstone of most SiLK analyses, and is the only SiLK tool currently providing data selection and filtering capabilities. While a new tool could be created for the specific purpose of MKset filtering, adding functionality to rwfilter serves to maintain the common flow of SiLK analyses and leverages the existing code base and implementation structure of the SiLK filter tool. In addition, the current SiLK MKset filter method uses rwfilter, and using a similar framework lends fairness to any comparison of our method to tuple filtering. To enable improved MKset filtering in the SiLK toolset, our design choice is to improve upon the MKset filter capability of SiLK's rwfilter tool.

4.2 Use Characteristics

In choosing a data structure for MKset filtering, the following algorithmic properties of such a filter were considered.

Insertion/Deletion

The MKset is loaded only once, with no insertions or deletions required during filtering. As such, for filtering purposes, structures with fast one-time allocation and insertion are preferable, with no considerations for deletion or modification of keys required.

Search

The number of search operations performed on the MKset will be as many as the number of input records to the filter, i.e., each SiLK record to be filtered will result in exactly one key search in the MKset. Combined with the one-time building phase, search speed is our most important feature of a MKset filter structure.

To illustrate the need for speed, one of the key advantages of Network Flow analysis over deep packet inspection methods (DPI) is improved ability to analyze data over a wide range of times and hosts, a trade-off for DPI's ability to determine exact host behaviour using actual application-level data. Our validation data set spans 6 months of flow collection on a small network of 567 IP addresses (two /24 networks and 60 externally routable addresses). Filtering this data for records matching unique source IP, destination IP, protocol (SIP, DIP, proto) pairs seen in a 2-week subset entails filtering about 140,000,000 flow records through an MKset of about 1,000,000 keys. Fast search speed is key to ensuring that analysts can perform such filtering on larger networks and longer time spans than this small example.

Memory

rwfilter's input records are processed as a stream. Consequently, the space requirements of the MKset hash table will be the largest contributor to the overall memory footprint of the filter tool. Our 534,354-key MKset from Section 4.2 is about 18MB in size, with each (SIP, DIP, proto) key requiring 33 bytes of storage (16 bytes for each IP address - allowing for both IPv4 and IPv6 addresses, and one for protocol) [3]. Even if we presume an overhead or loading factor that doubles this size, this MKset would still require less than 40MB of space when loaded into a search structure in memory. While memory usage remains pertinent to allow scaling to larger MKset searches, with current commodity computers commonly containing 4GB of RAM, memory usage is less important than speed considerations in our choice of data structure.

In summary, the ideal data structure for an MKset filter is one that can be built quickly and performs search operations as quickly as possible, with efficient memory usage as a secondary consideration.

4.3 Candidate Data Structures

With the use characteristics from Section 4.2 in mind, a desirable data structure for MKset filtering is one with the following properties, in order of importance:

- fast execution of search operations
- fast MKset creation
- efficient usage of memory

A case can be made for many different data structures to meet these requirements. We present analysis of three: splay trees, which provide fast search times by self-reorganization to optimize searches for common keys; minimal perfect hashing, which boast constant-time insertion and search operations with optimal space usage; and cuckoo hashing, which also give constant time insert and search operations.

SiLK's current MKset or "tuple" filter tool uses a red-black tree to search for keys in an MKset. With $O(\log n)$ insertion and search operations, and $O(n)$ memory usage, this provides our baseline for performance improvements.

Table 4.1 shows a summary of these search structures.

Structure	Insert	Search	Space
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(n)$
Splay Tree	$O(\log n)$	$O(\log n)$	$O(n)$
Minimal Perfect Hash	$O(1)$	$O(1)$	$O(n)$
Cuckoo Hash	$O(1)$	$O(1)$	$O(n)$

Table 4.1: Search Structure Summary

Splay Trees

Average-case lookup time may be improved with an adaptive data structure that rearranges itself based on search frequency of keys, such as a splay tree [30]. After each search operation, a splay tree rearranges entries such that frequently-searched keys are found more quickly by subsequent searches. As the filter progresses, searching the tree for frequently-occurring records in the input set would become faster.

Splay trees could be especially advantageous in a filter where, for example, one or more IP address fields have a bitmask modifier applied (i.e. matching on subnet inclusion), as many input SiLK records may map to the same search key. A splay tree has $O(\log n)$ amortized insert and search times, with actual times on homogenous input data often outperforming those of red-black trees [30]. However, on heterogeneous input data, splay trees display similar performance to red-black trees [30]. Considering that MKset filtering may be performed using any user-defined combination of flow fields, we can not easily predict the homogeneity on any particular combination of fields of the flow data to be filtered. As such, splay trees are not an ideal candidate for MKset filtering.

Minimal Perfect Hashing

A perfect hash function maps a set of N keys to M integers without collisions, where $M > N$. While memory usage is only a secondary consideration in our choice of data structure, the simplest perfect hash structures use $O(R)$ space, where R is the range of the input elements. With 16-byte IPv6 addresses as possible MKset keys, giving a range of 2^{16} for even a single-key MKset, simple perfect hashing is impractical for MKset filtering.

If $M = N$ in our definition of a perfect hash — that is, the size of the hash table is equal to the size of the input set — the function is called a minimal perfect hash (MPH) [32]. With constant time insertion and lookups, and optimal memory usage, data structures based on an MPH would match all of our criteria for improvement over red-black trees. MPH function generators exist for up to several billion keys, and McHugh et al. use MPH functions to store and query MAC addresses from network data [17]. However, MPH function generation time is at best linear in the number of keys [32], which would add an extra constant factor to the overall MKset creation time.

While perfect hash functions could be useful for filtering a large, known key set, the extra constant factor during building make perfect hash functions undesirable for a framework where flexibility and speed is desired in MKset generation and modification. Some manner of non-perfect hashing algorithm may be a good candidate to mitigate the tradeoff between space and time constraints over a variety of input sets. In addition, as implementation of an MPH algorithm is made difficult by the process of hash function generation, verification of correctness could be an obstacle to the using minimal perfect hashing for MKset filtering.

Cuckoo Hashing

The cuckoo hash [23] uses multiple hash functions to provide more than one possible spot for each key in the hash table. Using two functions h_1 and h_2 as an example, a key x is inserted by computing $h_1(x)$. If the space at $h_1(x)$ is occupied by a key y, then y is evicted to make room for x, and we place y in its alternate location $h_2(y)$, similarly evicting any key that may reside there. This process continues until a vacant location is found, or until K evictions have occurred, where K is the number of keys in the hash table. If K evictions occur before a vacant location is found, then the hash table is doubled in size and rebuilt. This insertion process succeeds in amortized constant time when amortized over the complete insertion process [23].

In our two-function example, searching for a key x in the hash table involves examining only two locations: $h_1(x)$ and $h_2(x)$, and so proceeds in constant time as well. Also, as shown by McHugh [15], 50% space utilization is guaranteed with two hash functions, and above 90% is guaranteed if we increase the number of hash functions to 4, giving a memory requirement of $O(n)$. In fact, McHugh's work [15] showed over 90% space utilization in practice even when the hash table is rebuilt after less than K evictions.

With constant-time insertion and lookup, $O(n)$ space, and linear time MKset modification operations, the cuckoo hash presents the combination of speed, memory, and data flexibility improvements over red-black trees that we desire. Our decision to use cuckoo hashing was further aided by the previous work done on cubags by McHugh et al., as the ability to leverage their work in creation and manipulation of cubags strengthen the case for using cuckoo hashing to attain our desired performance and functionality gains.

4.4 Cuckoo Hashing in SiLK

An MKset filter is performed as described by Algorithm 1. Figure 4.1 shows the rwfilter framework of Section 4.1 in which MKset filtering is implemented.

In our cubag model, the MKsetInsert and MKsetSearch functions in Algorithm 1 are cuckoo hash operations, and the modifyFields function applies cubag field modifiers such as AND/OR mask, reduce, right shift, and shift reduce, to a SiLK flow record, as detailed in Appendix A.

A cubag filter consists of reading a cubag file into a cuckoo hash, and, for each SiLK record to be filtered, applying any cubag field modifiers defined in the cubag file header and performing a cuckoo hash search to see if the SiLK record matches any keys in the cuckoo hash. If the SiLK record was found in the cuckoo hash, write it to the "pass" output stream specified by the user, otherwise write it to the "fail" output stream.

For a set of n SiLK input records and tuple file with k keys, k tuples must be inserted into the red-black tree at a cost of $O(\log k)$ per insertion, giving an initialization time of $O(k \log k)$. Each of the n SiLK input records must be searched for in the tree at $O(\log k)$ per search, for a total search time of $O(n \log k)$. This gives a total time for tuple filtering of $O(k \log k + n \log k)$. Replacing the $O(\log k)$ red-black tree insert and search times with $O(1)$ searches and insertions for cubag filtering, we get a total tuple filtering time of $O(k + n)$, and we see that cubag filtering can be expected to perform better asymptotically over tuple filtering.

The cubag filter module for rwfilter was designed to function as similarly as possible to the existing tuple architecture. This provides confidence that our tests are comparing the performance of the different data structures, rather than the details of the implementation framework.

Algorithm 1 MKset filter overview

```
Input: Set of SiLK data $INRECS$
    MKset file $MKSETFILE$
    SiLK fields $FIELDNAMES$ used to build MKset keys
    Field modifiers $FIELDMODS$ used to build MKset keys
    $PASSRECS$ : Output file to write SiLK records from $INRECS$ whose values in $FIELDNAMES$ matched a key in $MKSET$
    $FAILRECS$ : Output file to write all SiLK records from $INRECS$ whose values in $FIELDNAMES$ did not match a key in $MKSET$
Output: $PASSRECS$ output stream will contain SiLK records from $INRECS$ whose values in $FIELDNAMES$ matched a key in $MKSET$
    $FAILRECS$ output stream will contain SiLK records from $INRECS$ whose values in $FIELDNAMES$ did not match a key in $MKSET$
 1: boolean variable $PASS \leftarrow false$
 2: boolean variable $FAIL \leftarrow false$
 3: cuckoo hash $MKSET \leftarrow \emptyset$
 4: for every key $k\, in\, MKSETFILE$ do
 5:     MKsetInsert(MKSET, k)
 6: end for
 7: for every $rwrec\, in\, INRECS$ do
 8:     modifyFields(FIELDNAMES, FIELDMODS, rwrec)
 9:     $PASS \leftarrow$ MKsetSearch(MKSET, FIELDS, rwrec)
10:     if $PASS = true$ then
11:         $PASSRECS \leftarrow rwrec$
12:     else
13:         $FAILRECS \leftarrow rwrec$
14:     end if
15: end for
16: RETURN $PASSRECS, FAILRECS$
```

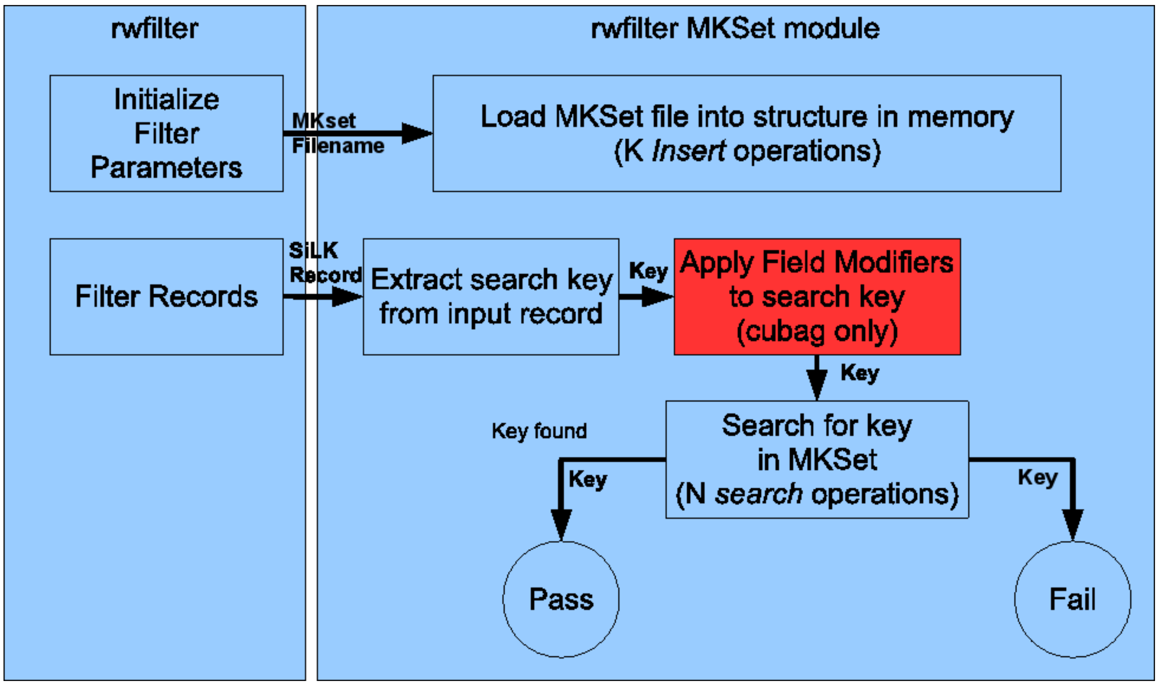
rwfilter
rwfilter MKSet module
Initialize Filter Parameters
MKset Filename
Load MKSet file into structure in memory (K *Insert* operations)
Filter Records
SiLK Record
Extract search key from input record
Key
Apply Field Modifiers to search key (cubag only)
Key
Search for key in MKSet (N *search* operations)
Key found
Key
Key
Pass
Fail

Figure 4.1: SiLK MKset Filter Framework

4.5 Implementation

With an existing framework for MKset filtering already present in the form of tuple filtering, major conceptual changes to the SiLK code were not needed. Rather, the challenge in implementation lay in adapting the cubag infrastructure, which was designed for building, modifying, and viewing cubag files, to fit into a model that includes filtering SiLK records with cubags similarly to tuple filtering.

4.5.1 Expanded Cubag Model

Our expanded cubag architecture was designed with the following considerations in mind:

- Common structures and functions should be available to all applications that work with cubags to create and manipulate cubag keys, and insert keys into a cuckoo hash structure
- Applications using cubags should be able to load keys from a cubag file into a cuckoo hash structure
- Applications using cubags should be able to search keys loaded from a cubag file for a given SiLK record, after applying any field modifiers that were applied during creation of the cubag file

4.5.2 Cubag Library

Cubag code is divided into two locations: the SiLK library called "cutab", and individual applications that use the cutab library. To implement our expanded cubag model as described, structures and functions to build, modify, read, and write cubags are located in the cutab library (see Appendix B). Code to perform user option processing such as input and output filenames and selection of SiLK fields and modifiers for cubag keys is left to the implementation of each individual application, such as cubag, cubagtool, and rwfilter.

The cutab library uses SiLK's Netflow data structures to perform cuckoo hash functions. The library could be extended to support other network data such as packet captures, or even generalized to use raw bit strings, by modifying its comparison and modification functions. This would be more than a trivial task; however, the main structure and framework of the library could provide a good starting point for general-purpose filtering of multi-dimensional data.

4.5.3 Cubag Filtering in rwfilter

With our expanded cubag code model, additions to rwfilter code to enable cubag filtering are focused on the logistics of adding the framework to enable the cubag filtering option. Using a framework similar to that used by SiLK's tuple filtering, the loading of cubag keys into a cuckoo hash, application of cubag field modifiers to SiLK records, and searching for SiLK records in the cuckoo hash, are performed through calls to the cutab library functions. We have thereby essentially replaced SiLK's red-black tree operations with similar operations using cuckoo hashing, with the caveat that we have also added the ability to perform cubag field modifications to SiLK records before filtering.

4.5.4 Code Changes

Our changes were made to SiLK version 2.3.1. The areas of code that required modification were in the rwfilter and cubag source, as well as the cutab library code. Our main contributions were to add a cuBagLoad() function to the cutab library and modify rwfilter to perform cubag filtering, and details can be found in Appendix B.

Chapter 5

Results

To validate our goal of improved performance in SiLK MKset filtering, we examine running times and peak memory usage for MKset filters using SiLK tuple filtering and our cubag filtering. We execute MKset filters using a range of input sets and MKset sizes, and we measure the performance effects of cubag field modification during cubag filtering by executing filters with the bit-masking modifier applied to the DIP field. Bit-masking is chosen as the SiLK field modifier because it is a common operation for subnet operations, a common operation to any networking task.

To validate our auxiliary goal of improved MKset creation over SiLK's tuple files using the cubag tool, we show the commands necessary for MKset generation, along with performance metrics for these commands.

All filters are timed using the GNU time tool [9], and peak memory usage is obtained with the Massif tool in the Valgrind software instrumentation framework [20]. All tests were performed on a quad-core, 3.1MHz computer running OpenSuSe Linux 11.1. SiLK release 2.3.1 [3], with McHugh et al.'s [15] cubag modifications, is the code base upon which our cubag filtering mechanism is built.

5.1 Data Set

Our data set consists of 142,143,967 SiLK flow records collected from a small hospitality network, representing 6 months of network activity. The internal network has 567 IPv4 addresses, broken down as follows:

- two /24 ranges (x.x.x.2-x.x.x.254) + 61 routable addresses
- 288 distinct internal source IP addresses (SIPs) seen
- 314 distinct internal destination IP addresses (DIPs) seen

- 3 internal SIPs not seen in internal DIPs (i.e. 3 internal hosts sent but did not receive packets)
- 29 internal DIPs not seen in internal SIPs (i.e. 29 internal addresses were sent packets but did not send any)

SiLK data sets of varying size were created by partitioning the full data set into subsets using SiLK's rwsplit tool. Each data set was then used to create MKset files with source IP, destination IP, and protocol (SIP, DIP, protocol) keys. Contiguous subsets of SiLK flow records were used to produce a consistent increase in MKset size. Each data set is labeled DS$< N >$, where N is the number of SiLK flow records in the set; the MKset created from a data set DS$< N >$ is labeled MK$< N >$.

The size and labels of the SiLK data sets and their resultant MKsets are shown in Table 5.1.

Data Set	#SiLK flows	MKset	#Keys
DS100M	100,000,000	MK100M	7,727,158
DS90M	90,000,000	MK90M	6,995,550
DS80M	80,000,000	MK80M	6,453,435
DS70M	70,000,000	MK70M	5,737,916
DS60M	60,000,000	MK60M	4,504,124
DS50M	50,000,000	MK50M	3,611,667
DS40M	40,000,000	MK40M	2,867,330
DS30M	30,000,000	MK30M	2,300,485
DS20M	20,000,000	MK20M	1,619,791
DS10M	10,000,000	MK10M	534,354

Table 5.1: Data Set and MKset Labels

5.2 Experimental Validation

To compare SiLK's tuple filtering performance to that of cubag filtering, two types of filtering tests are performed. In the first test, we filter a fixed number SiLK flows with each of our MKsets, to view the relationship between MKset size and filtering performance. In the second test, we filter a varying number of SiLK flows with a single MKset, to determine the effects of the number of input flows on an MKset filter. In addition, we present performance metrics measured during creation of the

MKset files, to verify that performance improvements in MKset filtering do not come at a cost of slower or less memory-efficient MKset creation.

5.2.1 MKset Creation

We begin by comparing the generation performance of three types of MKset files with keys SIP, DIP, and protocol: tuple files and cubag files both with and without the bit-masking cubag field modifier applied to the DIP field. We call these three types of files “tuple”, “cubag”, and “modcubag” files, and refer to tests involving these three file types as operating in “tuple mode”, “cubag mode”, and “modcubag mode”, respectively.

Modcubag mode differs from cubag mode in that the DIP field of each key is AND-bitmasked with the IP address 255.255.255.255 before insertion into the cubag. This trivial bitmask was chosen so that the effects of the field modification process could be measured while maintaining equal-sized MKsets for all three modes.

SiLK’s standard tool to create MKset files, the rwuniq tool, does not use the red-black tree structure. Instead, rwuniq uses a hash table to create a list of unique SiLK field values, which is output as ASCII text. As such, we expect that MKset creation times will be linear in the number of input records for both tuple and cubag modes, since the main difference between the two creation methods is utilization of different hashing techniques. We present performance metrics to illustrate the differences between the MKset creation methods. Since MKset creation performance is an aspect of previous work by McHugh et al. [15], our theoretical analysis of MKset creation performance is brief.

MKset Creation Commands

The following pipeline of Unix and SiLK commands are used to build a tuple file mkset.tuple.txt with fields SIP, DIP, and protocol, from a set of SiLK records in the file inputrecords.rwf:

```
rwuniq --fields=sip,dip,proto inputrecords.rwf | cut -f 1,2,3 -d
    "|" > mkset.tuple.txt
```

Our tests measure only the performance of the rwuniq tool, and not the cut command. Testing showed that the cut command added negligible time and no increase in peak memory to the overall creation tasks, so rwuniq was made the focus of these tests for simplicity's sake.

In contrast, the following commands create the MKset files mkset.cubag.cub and mkset.modcubag.cub used in cubag and modcubag modes:

Cubag:

```
cubag --set-file=mkset.cubag.cub:sip,dip,proto inputrecords.rwf
```

Modcubag:

```
cubag --set-file=mkset.cubag.cub:sip,dip,proto inputrecords.rwf
```

While not the main focus of our work, the simplicity of MKset creation with the cubag command relative to the tuple file creation pipeline adds value and usability to the MKset filtering process.

MKset Creation Performance: Tuple Mode vs Cubag Mode vs Modcubag Mode

As shown by Figure 5.1, MKset file creation was comparable in all three modes up to the point of 5.7M keys, with cubag creation time taking an average of 10% more time than tuple files. However, for larger sets, we see the cubag method perform more slowly in comparison to tuple, with cubag file creation time dropping to 23% less than tuple file creation at 7.7M keys. The cubag tool sorts keys before writing to disk, and the tuple files are not sorted. On all data sets, the times in Figure 5.1 would be smaller for cubag, and possibly faster than rwuniq, if cubag functioned similarly to rwuniq and did not sort the keys. To improve cubag generation times, this sorting step could be skipped. However, the sorting allows for stream-based cubag merge operations with the cubagtool utility program, and we choose to keep this step to allow the analyst the option to use this tool.

Examination of the hash table implementation used by SiLK's rwuniq tool for tuple creation indicates that this is likely due to a higher number of collisions during key insertion for the larger data sets. While cubag's cuckoo hashing code increases hash table size by a factor of two for unresolvable collisions, rwuniq's hash table adds blocks of a fixed size when the load factor exceeds 75%. As hash table size increases, this fixed block addition results in less reduction in load factor, and thus the number

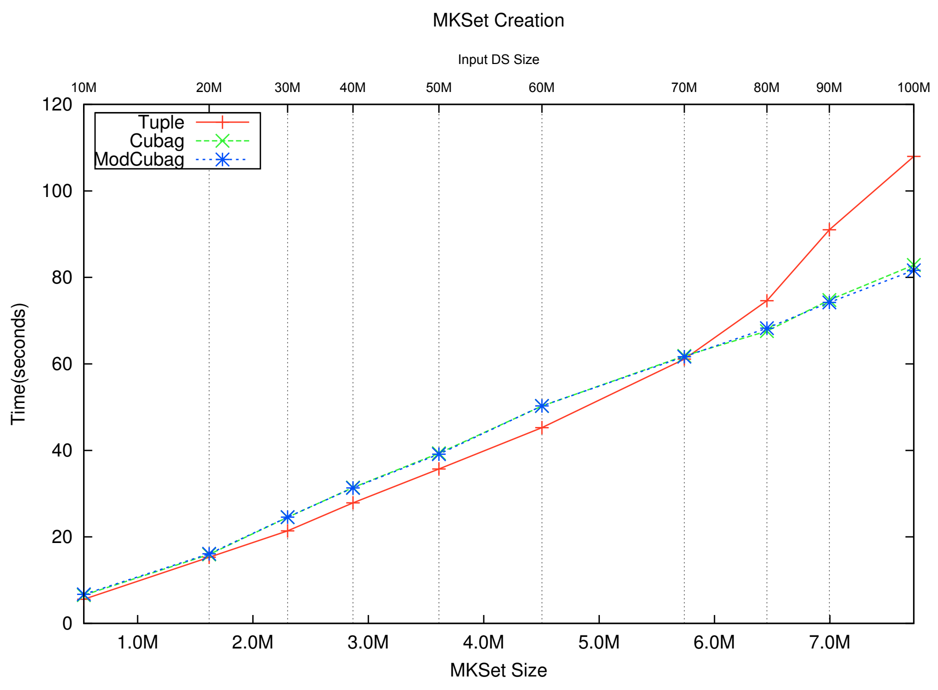

Figure 5.1: Time taken to build MKsets

of collisions and re-hashing increases. If memory usage were to become a limiting factor in MKset creation, analysis of actual hash table collisions in the tuple and cubag MKset creation processes should be a starting point for reduction in memory usage.

Figure 5.2 shows peak memory usage in the cubag and rwuniq tools during MKset file creation. We see a rough average of 33% decrease in memory usage in the cubag modes over tuple mode while generating the MKset file.

To explain the areas of constant memory usage in Figure 5.2, consider the following: the *load factor* of a hash table is defined as the ratio of occupied hash table locations to the total number of locations in the hash table. Except in perfect hashing [32], when a key is to be inserted into a hash table, there is some probability that its location in the table is already occupied, giving rise to a hash table collision. The SiLK hash table model maintains a load factor of less than 75% [3], after which a collision triggers a re-build of its hash table.

Cuckoo hashing with four hash functions, and McHugh's reduction in the number of evictions that cause a re-build of the hash table [15], can operate at a load factor of over 90% before collisions occur [23], and a hash table re-build only occurs when

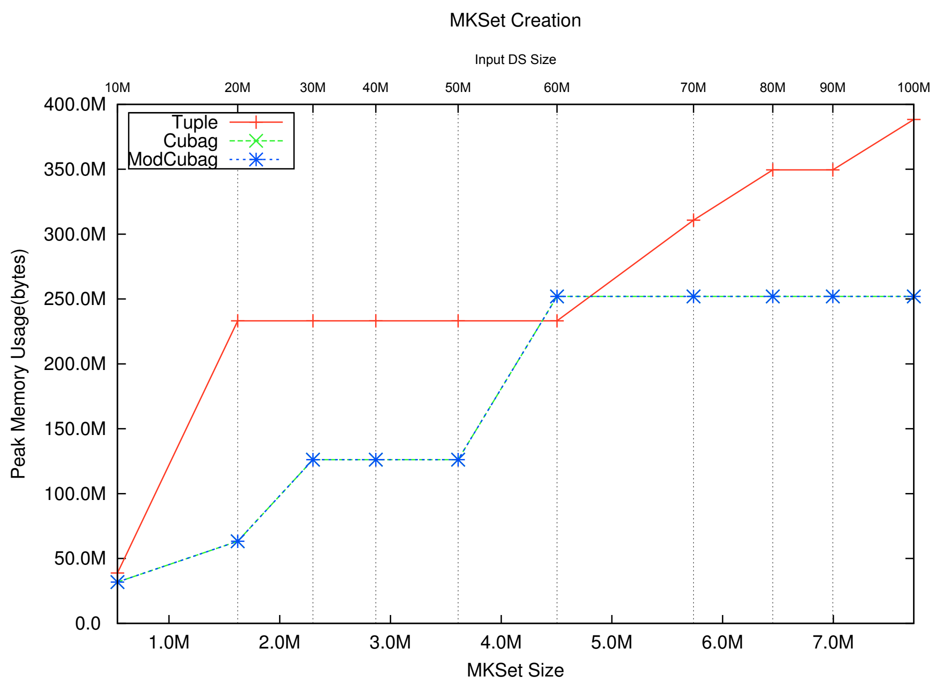

Figure 5.2: Peak memory usage while building MKsets

an unresolvable collision occurs. Increases in the number of hash table re-building operations for tuple MKset creation occur at MKset sizes of 1.6M, 5.7M, 6.5M, and 7.7M, these increases cause the rises in memory usage seen in Figure 5.2. In both cubag modes, these increases occur at 1.6M, 2.3M, and 4.5M. Between these increases in MKset size, hash table sizes remain constant while the numbers of keys in the tables increase. This is shown by the horizontal segments in Figure 5.2.

At 4.5M keys in Figure 5.2, we see memory usage for cubag modes surpassing that of tuple mode. This is explained by the fact that the cuckoo hash table has just increased in size, and the SiLK hash table will increase in size at the next MKset size increment.

These graphs indicate that the major preliminary step to an MKset filter, i.e. generating the MKset file, is performed comparably quickly, and with a smaller memory footprint by the cubag tool over SiLK's rwuniq, even with the added step of sorting the hash table to allow for cubag merging with cubagtool.

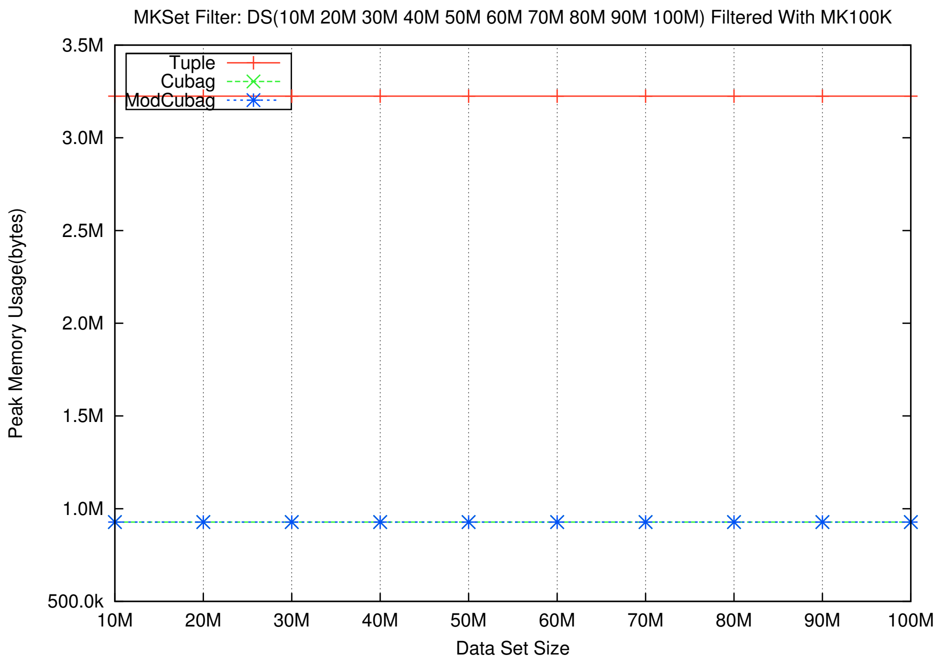

Figure 5.3: Peak memory usage by MKset filters with MK100K

5.2.2 MKset Filtering

Using our generated MKset files from Table 5.1, we perform two types of MKset filtering on our data sets. To view the relationship between MKset size and filtering performance, we filter DS100K with all MKsets. To determine the effects of the number of input flows on an MKset filter, we filter all data sets with MK100K, an MKset similar to those in previous tests as seen in Table 5.1, created from 100,000 SiLK records and containing a total of 6,207 keys. The size of this MKset was chosen to be small enough that the time taken reading and building the MKset would not dominate the time taken to filter each data set with the MKset, but large enough to produce measurable times for filtering operations.

Variable Input size, Fixed MKset Size: All Data Sets Filtered with MK100K

Figures 5.3 and 5.4, and Tables 5.2 and 5.3 show performance metrics from our variable input size filters. Using MKset MK100K, (which contains 6,207 keys), MKset filtering performance was measured over all input sizes.

From Figure 5.3 we see that input set size has no effect on peak memory usage, in both modified and unmodified MKset filters.

Data set	Tuple Filter Memory	Cubag Filter Memory	Modcubag Filter Memory
10M	3224688	927760	927784
20M	3224688	927760	927784
30M	3224688	927760	927784
40M	3224688	927760	927784
50M	3224688	927760	927784
60M	3224688	927760	927784
70M	3224688	927760	927784
80M	3224688	927760	927784
90M	3224688	927760	927784
100M	3224688	927760	927784

Table 5.2: Peak memory usage by MKset filters with MK100K

Data Set	Tuple Filter Time	Cubag Filter Time	Modcubag Filter Time
10M	11.07	5.82	6.11
20M	22.65	11.83	11.72
30M	33.91	17.33	17.35
40M	43.69	22.88	23.08
50M	53.33	28.72	29.08
60M	64.21	34.84	33.96
70M	72.72	40.27	39.47
80M	81.91	44.86	44.74
90M	90.89	49.94	49.40
100M	99.29	54.57	54.68

Table 5.3: Time taken by MKset filters with MK100K

Recall from Chapter 4 that we expect to see $O(1)$ search time per input record for cubag filters, and $O(\log k)$ for tuple filters, with k the number of keys in the cubag or tuple file, respectively. Total filter time, then, is expected to be $O(n)$ for cubag filtering, and $O(n \log k)$ for tuple filters. However, with k held constant for the variable input size, fixed MKset size tests summarized in Figure 5.4, we expect overall filtering time to be linear in the number of input records for both cubag and tuple modes. R^2 tests on the data as seen in table 5.4 confirm that filter time is linear in

Mode	Function	R^2 Value
Tuple	$y = 9.7553x + 3.7127$	0.99781
Cubag	$y = 5.4634x + 1.0525$	0.9984
Modcubag	$y = 5.4054x + 1.2293$	0.99916

Table 5.4: Regression Analysis for MKset filtering with MK100K

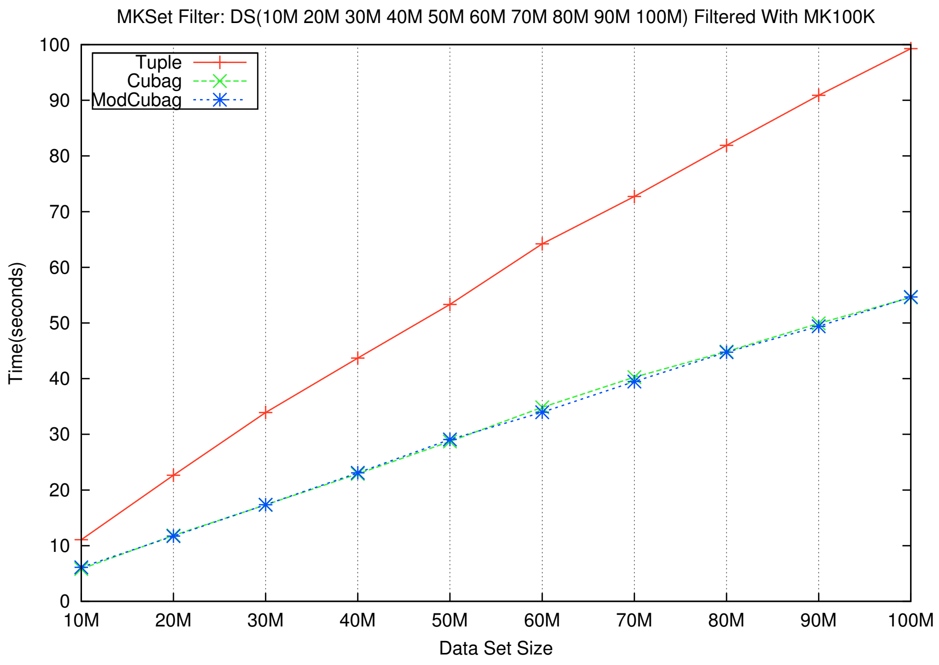

Figure 5.4: Time taken by MKset filters with MK100K

n, with tuple mode incurring a larger constant multiplier. Figure 5.6 serves as simple confirmation that execution time of cubag filtering is lower than that of tuple filtering over a range of input sizes.

Fixed Input Size, Variable MKset Size: DS100K Filtered With All MKsets

Figures 5.5 and 5.6, and Tables 5.5 and 5.7 show performance measurements from our variable MKset filters. Keeping input size constant at 100,000 records, MKset filtering performance was measured over all MKset sizes, with and without field modifiers.

R^2 tests on data as seen in Table 5.6 reveals that memory usage by rwfilter during MKset filtering is linear in the size of the MKset, which is expected since the main contributors to memory usage in MKset filters of using MKsets of size k keys are the red-black tree and the cuckoo hash table, each taking $O(k)$ space. Also, we see a negligible increase in memory demands for the bit-masking modifier of 104 bytes for every MKset filter, to store the modifier value and some structure overhead.

Table 5.5 shows that the cuckoo hash tables consume 50% or less memory than the red-black trees. Consideration of the red-black tree algorithm and inspection of the SiLK implementation reveal that two child node pointers per MKset key can be

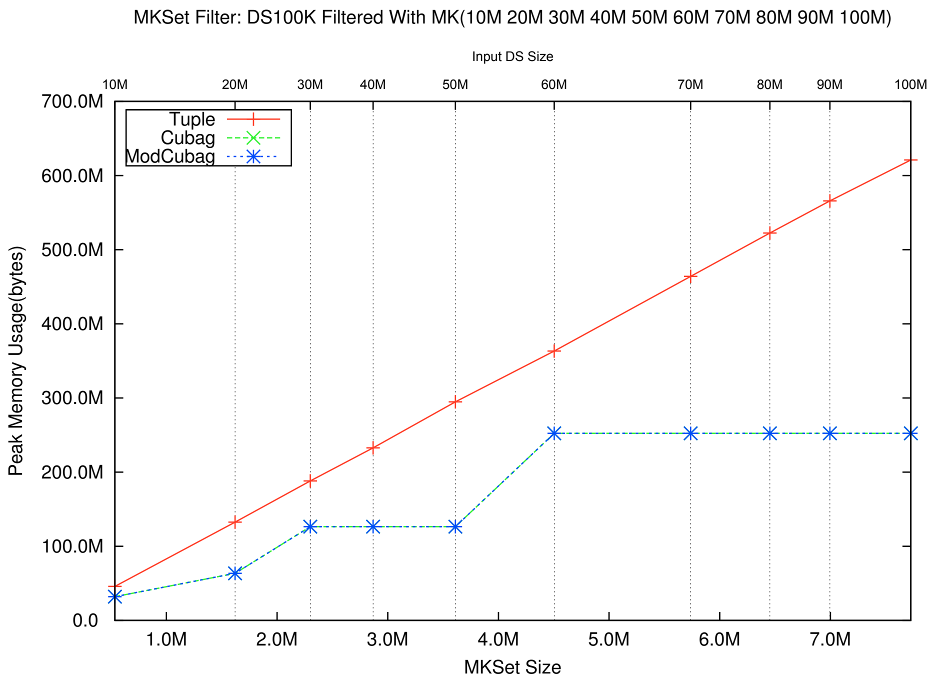

Figure 5.5: Peak memory usage by MKset filters on DS100K

attributed to this memory increase.

With an average time decrease of 90%, cubag's cuckoo hash structure has outperformed SiLK's red-black tree in these MKset filters, as shown in Figure 5.6 and Table 5.7. With an added .01% in filtering time, adding the bit-masking field modifier to the cubag filters has a negligible impact on speed.

In addition, we see in Figure 5.6 that filter time is constant per input record for the cubag filters, while time increases linearly in MKset size for the tuple filters.

Given a fixed input set, the cubag filters improve time and memory usage in MKset filters over SiLK's tuple filtering tool. As well, bit-mask field modification at filter time adds negligible reduction in execution speed.

5.2.3 Summary

Analyses of our experimental validation tests have shown comparable time and a 33% reduction in memory usage for MKset creation using cubags instead of SiLK tuple files. MKset filtering using a cubag of size k shows $O(\log k)$ improvement in execution time over tuple filtering using a tuple file of size k, with a 50% reduction in memory usage.

MKset Size	Tuple Filter Memory	Cubag Filter Memory	Modcubag Filter Memory
534354	45,877,424	32,093,976	32,094,080
1619791	132,582,008	63,551,080	63,551,184
2300485	188,045,352	126,465,936	126,466,040
2867330	232,817,960	126,465,488	126,465,592
3611667	294,757,776	126,465,312	126,465,416
4504124	363,224,112	252,295,168	252,295,272
5737916	464,084,320	252,294,704	252,294,808
6453435	522,477,072	252,294,360	252,294,464
6995550	565,890,104	252,294,976	252,295,080
7727158	621,027,464	252,295,080	252,295,184

Table 5.5: Memory Footprint of MKset filters on DS100K

Mode	Function	R^2 Value
Tuple	$y = 60,000,000x + 10,000,000$	0.99425
Cubag	$y = 30,000,000x + 30,000,000$	0.85455
Modcubag	$y = 30,000,000x + 30,000,000$	0.85455

Table 5.6: Regression Analysis for MKset filtering on MK100K

MKset Size	Tuple Filter	Cubag Filter	%Impr	Modcubag Filter	%Impr
534354	6.86	0.70	89.80	0.72	30.04
1619791	19.99	1.90	90.50	1.95	52.07
2300485	28.54	3.14	89.00	3.19	32.75
2867330	36.06	3.56	90.13	3.50	45.68
3611667	45.66	4.22	90.76	4.17	57.10
4504124	57.89	6.27	89.20	6.26	30.54
5737916	74.27	7.45	90.00	7.44	45.64
6453435	84.47	8.09	89.97	7.87	51.71
6995550	91.40	8.51	90.42	8.28	55.42
7727158	100.77	9.19	90.89	9.18	59.37

Table 5.7: Time taken by MKset filters on DS100K

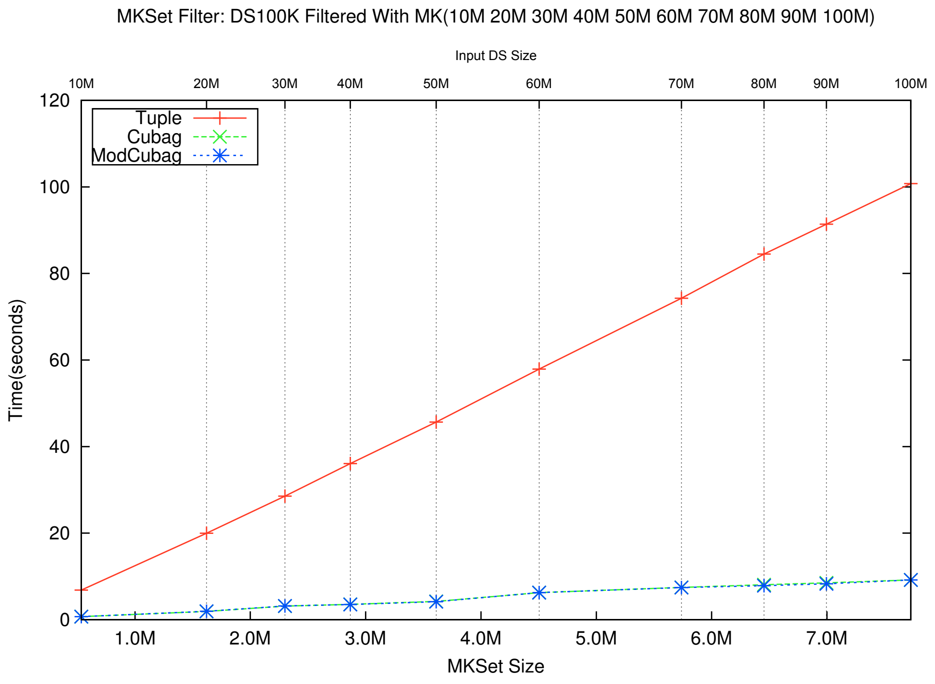

Figure 5.6: Time taken by MKset filters on DS100K

Chapter 6

Conclusions

Network flow analysis is used by network analysts to monitor [34], plan, and maintain [25] IP networks. One common network flow analysis task is MKset filtering, which involves filtering flow records based on inclusion in a “multi-key set”, or MKset. An MKset is a set containing one or more unique “multi-key” elements, each of which consists of values for one or more network flow attributes. The SiLK suite of network flow analysis tools provides a method of performing MKset filtering called “tuple filtering”, which uses red-black trees [6] to speed up the filtering process.

6.1 Main Goals

Our main goals for improving MKset filtering in SiLK were as follows:

- faster run times
- decreased memory usage
- added functionality for manipulation of set keys

Using the fast and memory-efficient cuckoo hashing algorithm, we have improved execution time for MKset filtering in SiLK from $O(n \log k + k \log k)$ to $O(n + k)$, where n is the number of SiLK records to be filtered, and k is the number of keys in the MKset. Experimental results confirmed $O(\log k)$ speed improvement and two-fold memory efficiency in cubag filtering as compared to tuple filtering.

Previous work by McHugh et al. [15] provided a cuckoo-hash-based MKset structure called a cubag. Building upon this work, we also have added cubag key modification at MKset filter time with negligible effects on performance compared to filtering without key modification.

6.2 Auxiliary Goals

Auxiliary goals for our cubag filters were:

- ensure any new tools or methods fit within the standard flow of the SiLK tools
- maintain or improve the number of commands and non-SiLK scripting required to generate, store, and filter MKsets
- keep in mind that we may want to modify or merge MKsets to allow greater filtering flexibility, similarly to SiLK's current bag and set tools

By fitting our cubag filter mechanism into a similar framework as SiLK's tuple filters, we have created a tool that fits with the SiLK tool paradigm for tool usage and code structure. As shown in the results section, creation, storage, and manipulation of cubags via the existing suite of cubag tools uses less commands than piping rwuniq output through scripts to create text-based tuple files, for which no manipulation tool exists.

With faster execution times and less space usage than tuple filtering, and the added functionality of cubag field modification, we have created an improved tool to replace SiLK's tuple filtering, through modification of the SiLK tools code to allow MKset filtering with cuckoo bags using cuckoo hashing. We have provided faster and more flexible filtering to network analysts, which allows the analysts to explore their network traffic data in a more iterative and flexible manner.

6.3 Future Work

The next step in our work will be to integrate the cubag suite of tools, including our rwfiltercubag, into the main SiLK code repository. With better performance, easier usage, and flexible file management, the cubag suite will give SiLK analysts the power to perform larger and more complex analyses than previously possible.

While cubag filtering provided considerable speed and memory improvements over tuple filters, potential improvements in MKset filtering speed remain, ranging from minor tweaks to major changes.

For filtering tasks involving large MKsets relative to the number of SiLK records to be filtered, an improved filtering model could involve reading subsets of an MKset file into memory as necessary. For instance, if an MKset was constructed with a large number of hosts on the Internet known to be involved in botnet activity and the port numbers on which they undertake this activity, an analyst may want to search in targeted time periods, such as a few hours, for potential botnet activity on their internal network during those times, by performing an MKset filter on SiLK flow records for the given time period with the botnet MKset. If the IP addresses seen in a small set of SiLK records are few and closely related, a sorting of the k keys in the MKset, say with the $O(k \log k)$ merge sort [11] algorithm, will tend to place keys matching this small set of SiLK records closely together on disk in the MKset file. Taking advantage of the sorted MKset keys, a targeted subset of the MKset could be read from disk that is likely to contain most or all of the keys that could match the SiLK records to be filtered. If this botnet search is a commonly-performed analysis, this subset-loading algorithm could provide gains in filtering time that outweigh the $O(k \log k)$ sorting of the botnet MKset.

Currently, cubags are stored by writing hash keys to simple binary flat files. The loading speed of a cubag file into a hash table during the filter initialization stage could be improved by implementing a file format that is a serialized version of the hash table, with actual memory offsets and hash table specifics stored in the cubag file. Ensuring successful deserialization on different architectures with different endianness and word sizes would require careful implementation of such a file format, and with $O(1)$ cuckoo hash amortized insertion time, it is expected that this would result in only small constant time gains; however, as cubag sizes scale upward, this could be a worthwhile improvement.

Before loading an input file, SiLK's rwfilter code performs initial tests on the file header to possibly fail the entire file before any records are checked. For instance, if the filter parameters include an IPv6 address, and the input file header specifies that its records are IPv4 only, then rwfilter can simply write all the records in that file to its "fail" output stream. Adding the cubag filter parameters into this checking process should be done to maintain consistency with the rwfilter framework when our code is included in the main SiLK repository.

For further clarification, note that these header checks are more relevant when using rwfilter's file selection switches, and not just specifying a single input file as we have done in our examples.

Neither tuple nor cubag filtering currently take advantage of rwfilter's multi-threading capabilities. Enabling multiple concurrent search threads, during either kind of MKset filter, should result in an improvement in filtering speed on multiple-core systems.

MKset filtering could be extended to other problem domains by modification of the data structures and functions in the cutab library. By modifying cutab to be a general-purpose API for cuckoo hashing, applications for tasks such as data mining, classification, and database querying could benefit from the concepts employed by our improved tuple filtering in SiLK.

Bibliography

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] B. Bruins and D. Kerr. Network flow data export, Oct 2001. US Patent 6243667.

[3] CERT/NetSA at Carnegie Mellon University. SiLK (System for Internet-Level Knowledge). `http://tools.netsa.cert.org/silk/index.html`. [Accessed: July 30, 2010].

[4] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[5] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 13, pages 273–301. MIT Press and McGraw-Hill, second edition, 2001.

[7] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 137–148, New York, NY, USA, 2003. ACM.

[8] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[9] Free Software Foundation. GNU Time command. `http://www.gnu.org/software/time/index.html`, 1998.

[10] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. *SIGCOMM Computer Communications Review*, 35(4):229–240, 2005.

[11] M. Kronrod. Optimal ordering algorithm without operational field. *Soviet Mathematics - Doklady*, (10):744–746, 1969.

[12] A. Kumar and S. Bhatia. Scalable flow analysis. In *Proceedings of FloCon*, 2006. `http://www.cert.org/flocon/2006/presentations/scalable\_flow\_analysis2006.pdf`.

[13] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 217–228, New York, NY, USA, 2005. ACM.

[14] X. Li, F. Bian, H. Zhang, C. Diot, R. Govindan, W. Hong, and G. Iannaccone. Mind: A distributed multi-dimensional indexing system for network diagnosis. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications*, pages 1 –12, Apr 2006.

[15] J. McHugh. Sets, bags and rock and roll; analyzing large sets of network data. In *Proceedings of ESORICS 2004*, LNCS, pages 407–422. Springer, 2004.

[16] J. McHugh, R. McLeod, and V. Nagaonkar. Passive network forensics: behavioural classification of network hosts based on connection patterns. *SIGOPS Operating Systems Review*, 42(3):99–111, 2008.

[17] J. McHugh and S. Parikh. Flow analysis in a wireless environment with short DHCP leases. In *Proceedings of FloCon*, 2008. `http://www.cert.org/flocon/2008/presentations/FloCon08-mchugh-sanket.pdf`.

[18] A. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[19] K. Nagaraj, K. Naidu, R. Rastogi, and S. Satkin. Efficient aggregate computation over data streams. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1382–1384, Washington, DC, USA, 2008. IEEE Computer Society.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[21] B. Nickless. Combining cisco netflow exports with relational database technology for usage statistics, intrusion detection, and network forensics. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 285–290, Berkeley, CA, USA, 2000. USENIX Association.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[23] R. Pagh and F. Rodler. Cuckoo hashing. In *Algorithms ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, chapter 10, pages 121–133. Springer Berlin Heidelberg, Berlin, Heidelberg, August 2001.

[24] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sflow: A method for monitoring traffic in switched and routed networks. RFC 3176, Internet Engineering Task Force, September 2001.

[25] A. Pras, R. Ramin, A. Sperotto, T. Fioreze, D. Hausheer, and J. Schoenwalder. Using netflow/ipfix for network management. *Journal of Network and Systems Management*, 17:482–487, 2009. 10.1007/s10922-009-9138-0.

[26] LLC Qosient. Argus flow. `http://www.qosient.com/argus/index.shtml`, Aug 2010.

[27] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, 2004. `http://berkeley.intel-research.net/sylvia/pht.pdf`.

[28] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329 – 350, 2001.

[30] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[31] J. Sommers, P. Barford, N. Duffield, and A. Ron. Accurate and efficient SLA compliance monitoring. In *SIGCOMM '07: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 109–120, New York, NY, USA, 2007. ACM.

[32] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Commununications of the ACM*, 20(11):841–850, 1977.

[33] S. Wang, R. State, M. Ourdane, and T. Engel. Flowrank: ranking netflow records. In *IWCMC '10: Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, pages 484–488, New York, NY, USA, 2010. ACM.

[34] X. Yin, W. Yurcik, M. Treaster, Y. Li, and K. Lakkaraju. Visflowconnect: Netflow visualizations of link relationships for security situational awareness. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, pages 26–34, New York, NY, USA, 2004. ACM.

[35] W. Yurcik and Y. Li. Internet security visualization case study: Instrumenting a network for netflow security visualization tools. In *In 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[36] C. Zhang and A. Krishnamurthy. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report Tr-703-04, Princeton University, 2004.

Appendix A

Cubag Syntax

The following syntax description comes from the John McHugh's cubag manual [15].

```
syntax
cubag --bag-file=OUTPUTFILE:keyfield[(modifier)][,keyfield...]
:datafield[,datafield ...]

Field names are not case sensitive and names can be abbreviated
to the shortest unambiguous string. The following fields can be
used as keys:

sIP, 1 :
Source IP
dIP, 2 :
Destination IP
sPort, 3 :
Source Port - if ports are specified as keys zero values
will be supplied for protocols other than TCP and UDP.
dPort, 4 :
Destination Port
Protocol, 5 :
Protocol
Packets, pkts, 6 :
Packet count
Bytes, 7 :
Byte count
Flags, 8 :
```

TCP Flags - set to zero for non-TCP flows

sTime, 9 :
Start time
Duration, 10 :
Duration
eTime, 11 :
End time
Sensor, 12 :
Sensor index
In, 13 :
Input interface index
Out, 14 :
Output interface index
nhIP, 15 :
Next Hop IP
InitialFlags, 26 :
Flags from the first packet of a TCP session. Zero if not
a TCP packet or if not collected.
SessionFlags, 27 :
OR of the flags from the remainder of the session. Zero if
not a TCP packet or if not collected.
Attributes, 28 :
Miscellaneous information about the flow including its
reason for termination. Bit map with 8 bits.
0x01 :
Additional TCP-state machine information is available.
This must be set for every TCP flow where the init_flags
and rest_flags fields are set.

0x08 :
Flow received packets following the FIN packet that were
not ACK or RST packets.
0x20 :
Flow ends prematurely due to a timeout by the collector.
0x40 :
Flow is a continuation of a previous flow that was killed
prematurely due to a timeout by the collector.
0x80 :
Used as a flag to mark a record as having IPv6 addresses.
Application, 29 :
Canonical port for a limited set of applications.
TypeClass :
An 8 bit field that is interpreted with respect to the
silk.conf file for the data in question.
sTime+msec, 22 :
Start time plus milliseconds. Only the seconds value is used
as a key. The milliseconds are discarded. Included for
completeness only.
eTime+msec, 23 :
End time as above.
Dur+msec, 24 :
Duration, as above.
icmpTypeCode, 25 :
256 * ICMP Message Type + ICMP Message Code. Set to (255,255),
an illegal value, if the protocol is not ICMP.
scc, 18 :
Source country code, a two character code associated with the
country of the source IP.
dcc, 19 :
Destination country code, a two character code associated with
the country of the destination IP.

```
IPversion :
This is a 1 byte key field that will be included automatically
when mixed IPv4, IPv6 addresses are possible and an IP address
is specified as a key field.

keyfield modifiers
Some key fields can be operated on in a number of ways.
Fields that contain IP addresses can be masked.  IPv4
fields can be divided or reduced. Time, duration, and
volume fields (used as keys) can be divided or reduced.
TCP flag fields can be masked. This allows, for example,
removing the significance of the non-state flags so that
flows containing, for example ACK and PSH,ACK would have
the same key field. At the present time, most 8 and 16
bit fields cannot be modified.  The following operations
are currently supported:

& AND mask
Applies to IP fields and TCP flag fields only.  The modifier is a
parseable IP address for V4 and V6 IP fields, an integer for V4 IP
and flag fields, and a string containing any combination of the
characters "FSRPAUEC" for flag fields.  The key field is "ANDed"
with the mask value before insertion into the key. Masking the low
order bits of an IP address with 0s has the effect of aggregating
volumes by subnet.

| OR mask
Similar to AND but the operation is a bitwise OR.

/ divide
Applies to time, duration, and volume key fields. The key field value
is integer divided by the operand before insertion in the key.  This
```

could be used to change the units of a field, e.g. duration divided by 60 gives minutes. Note that rounding does not take place. This should not matter for a key field.

/* reduce
Similar to divide, but multiplies the key field back to its original scale after division. Applied to epoch times, this allows time printing routines to treat the key field as a normal time_t, but with reduced precision.

>> right shift
Applies to IP addresses. Shifts bits right by the specified amount, effectively leaving a subnetwork prefix.

>< shift reduce
Applies to IP addresses. Shifts bits right by the specified amount, then left by the same amount effectively leaving a CIDR specification

Appendix B

Code Changes/Additions

cutab.c

cuBagLoad() As the cubag creation, modification, and viewing tools do not require loading of a full cubag file from disk into memory, there was no function written in the cubag library to do so. A new function, cuBagLoad(), was created to complete the cubag suite's shared library of functions to work with cuckoo hash tables (cutab.c). Some modification to the existing cuBagReadHeader() function was necessary to read all of the required header information, but the bulk of the work here entailed meshing portions of the cubag tool's hash creation functions, the SiLK stream I
O library, and some new code to calculate the optimal hash table size based on the number of entries in the cubag.

rwfiltercubag.c

Setup/Cleanup Some standard SiLK tool functions are used to handle input options, provide a hook to register rwfiltercubag with the main rwfilter code, and specify startup and teardown operations for the specified cubag file. In particular, the setup function calls the new cuBagLoad() to get header information and read the cubag into a cuckoo hash, and the teardown function calls cutab.c's cuBagFree() function to release any memory allocated for the cubag structures.

cuBagCheck() The meat of the new functionality lies in cuBagCheck(), which is the function that gets called by rwfilter to test a SiLK record for inclusion in the specified cubag file. The flow is as follows:

- copy the cubag key fields, as specified in the cubag header, into a buffer to be passed to cutab.c's cuBagLookup()

- apply any key modifiers to the buffer's keys, again as specified by the cubag header
- call cutab.c::cuBagLookup() to test whether the given SiLK record matches a key in the cuckoo hash built from the cubag file
- return pass or fail to rwfilter

rwfilter.c

Some minor modifcations were necessary to include the rwfiltercubag.c functionality in rwfilter's filter options.