Carnegie Mellon University
Software Engineering Institute

Network Traffic Analysis with SiLK

Analyst's Handbook for SiLK Version 3.15.0 and Later

AUGUST 2020

Paul Krystosek
Nancy Ott
Geoffrey Sanders
Timothy Shimeall

CERT® Situational Awareness Group

Carnegie Mellon University
Software Engineering Institute

Network Traffic Analysis with SiLK

Analyst's Handbook for SiLK Versions 3.15.0 and Later

Paul Krystosek
Nancy M. Ott
Geoffrey Sanders
Timothy Shimeall

August 2020

CERT® Situational Awareness Group

https://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

DM20-0675

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

List of Examples

List of Hints

This page intentionally left blank.

Acknowledgements

The authors wish to acknowledge the valuable contributions of all members of the CERT® Situational Awareness group and the CERT Engineering Group, past and present, to the concept and execution of the SiLK tool suite and to this handbook. Many individuals served as contributors, reviewers, and evaluators of the material in this handbook.

The authors also gratefully acknowledge the many SiLK users who have contributed immensely to the evolution of the tool suite.

Lastly, the authors wish to acknowledge their ongoing debt to the memory of Suresh L. Konda, PhD, who led the initial concept and development of the SiLK tool suite as a means of gaining network situational awareness.

This page intentionally left blank.

Goals for Network Traffic Analysis with SiLK

How to Use This Handbook

Network Traffic Analysis with SiLK: Analyst's Handbook for SiLK Versions 3.15.0 and Later (also known as the *SiLK Analyst's Handbook*) is an introduction to methods of analyzing network traffic, illustrated by commands from the SiLK tool suite. The focus is on learning to identify traffic features important to the security of information on the network. The handbook moves from a basic understanding of network flow and the SiLK tool suite through a series of examples that illustrate how to use SiLK to analyze network behavior.

The examples in this handbook are mainly command sequences that illustrate specific analysis concepts. Examples are commonly discussed on a line-by-line basis in the text and presented as command and output listings. In general, examples are also associated with a specific task (or tasks), indicated in the section and in the example caption. Case studies take a deeper dive into specific topics for analysis.

For readers already familiar with SiLK, the explanations of SiLK commands in the text of this handbook are kept short enough not to be redundant. More complete discussions of the commands and their parameters are provided in the *SiLK Reference Guide* and the `man` pages for the SiLK commands. Readers who are interested in analyzing network flow records with other tools than SiLK are encouraged to read the overall description of the analysis approaches, then use the description of commands to find parallels using the tool suite of their choice.

How This Handbook Is Organized

This handbook contains the following chapters:

1. **Introduction to SiLK** provides a short overview of some of the background necessary to begin using the SiLK tools for analysis. It includes a brief introduction to the SiLK suite and describes the basics of network flow capture by sensors and storage in the SiLK flow repository. It also discusses the analysis process used in this handbook and its application to situational awareness, incident response, and threat hunting. Finally, it describes the dataset used for the examples in this handbook.

2. **Basic Single-path Analysis with SiLK: Profiling and Reacting** describes the most straightforward analysis approach and applies it to several example analyses. It introduces some of the core SiLK commands and uses them to analyze network traffic.

3. **Case Studies: Basic Single-path Analysis** applies the single-path analysis approach to several extended examples, focusing on how those examples were developed from an initial problem statement through executable commands.

4. **Intermediate Multi-path Analysis with SiLK: Explaining and Investigating** explains a more complex, intermediate form of analysis which applies basic, single-path analysis in a multi-pronged structure. The chapter describes how multi-path analysis can be applied and includes a fuller exploration of SiLK tools that may be useful for this type of analysis.

5. **Case Studies: Intermediate Multi-path Analysis** applies multi-path analysis to extended examples.

6. **Advanced Exploratory Analysis with SiLK: Exploring and Hunting** discusses the use of SiLK to deal with open-ended, often iterative analyses that incorporate both single-path and multi-path methods. It also describes more sophisticated uses of the SiLK tool suite that support complex analyses of network behavior.

7. **Case Studies: Advanced Exploratory Analysis** applies exploratory analysis to an extended example.

8. **Extending the Reach of SiLK with PySiLK** describes how to extend the functionality of the SiLK tool suite by using the Python scripting language.

9. **Tuning SiLK for Improved Performance** discusses techniques to improve the efficiency and performance of SiLK when working with very large datasets.

The appendices to this guide provide information about common network protocols, list useful Unix commands, and give sources for additional information about the SiLK tool suite and network analysis.

What This Handbook Doesn't Cover

This handbook does not contain an exhaustive description of all the tools in the SiLK tool suite or of all the options in the described commands. Rather, it offers concepts and examples to allow analysts to accomplish needed work while continuing to build their skills and familiarity with SiLK.

- Every SiLK tool includes a `--help` option that briefly describes the command and lists its parameters.
- Every tool also has a manual page (also called a `man` page) that provides detailed information about the use of the tool. These pages may be available on your system by typing `man` *command*. For example, type `man rwfilter` to see information about the `rwfilter` command.
- The SiLK Documentation page at https://tools.netsa.cert.org/silk/docs.html includes links to individual manual pages.
- The *SiLK Reference Guide* is a single document that bundles all of the SiLK manual pages. It is available in HTML and PDF formats on the SiLK Documentation page (https://tools.netsa.cert.org/silk/docs.html).

This handbook deals solely with the analysis of network flow record data using an existing installation of the SiLK tool suite. For information on installing and configuring a new SiLK tool setup and on the collection of network flow records for use in these analyses, see the "Installation Information" section of the SiLK Documentation page at https://tools.netsa.cert.org/silk/docs.html#installation.

Chapter 1

Introduction to SiLK

Network analysts need to build an ongoing perspective on the traffic passing over their networks. This perspective is often built on information about the traffic (such as volumes, timing, and communication paths), rather than on the traffic itself. This chapter introduces the tools and techniques used to store such information, particularly in the form known as *network flow*. It will help you to become familiar with the structure of network flow data, how the SiLK collection system gathers those data from sensors, and how to use those data.

Upon completion of this chapter you will be able to

- describe a network flow record and the conditions under which the collection of one begins and ends
- describe the types of SiLK flow records
- describe the structure of the SiLK flow repository
- understand the steps involved in analyzing network flow data
- describe common applications of the analysis process
- describe the dataset for the examples in this handbook

1.1 What is SiLK?

The System for internet-Level Knowledge[1] (SiLK) tool suite is a highly scalable flow-data capture and analysis system developed by the CERT Situational Awareness group at Carnegie Mellon University's Software Engineering Institute (SEI). The SiLK tools provide network security analysts with the means to understand, query, and summarize both recent and historical traffic data represented as network flow records (also referred to as "network flow" or "network flow data" and occasionally just "flow"). These tools provide network security analysts with a relatively complete high-level view of traffic across an enterprise network, subject to placement of sensors.

Analyses using the SiLK tools provide insight into various aspects of network behavior. Some example applications of this tool suite include:

[1] The suite name, and in particular the capitalization, were chosen in memory of Dr. Suresh L. Konda, who was the inspirational leader for the creation of the initial suite prior to his sudden passing.

- supporting network forensics: identifying artifacts of intrusions, vulnerability exploits, worm behavior, etc.
- providing service inventories for large and dynamic networks (on the order of a /8 Classless Inter-Domain Routing (CIDR) block)
- generating profiles of network usage (bandwidth consumption) based on protocols and common communication patterns
- enabling non-signature-based scan detection and worm detection, for detection of limited-release malicious software and for identification of precursors

These examples, and others, are explained further in this handbook. By providing a common basis for these analyses, the SiLK tools provide a framework for developing network situational awareness.

Common questions addressed via flow analyses include (but aren't limited to)

- What is on my network?
- What constitutes typical network behavior?
- What happened before, during, and after an event?
- Where are policy violations occurring?
- Which are the most popular web servers?
- How much volume would be reduced by applying a blacklist?
- Do my users browse to known infected web servers?
- Is a spammer on my network?
- When did my web server stop responding to queries?
- Is my organization routing undesired traffic?
- Who uses my public Domain Name System (DNS) server?

1.2 The SiLK Flow Repository

1.2.1 What is Network Flow Data?

NetFlow is a traffic-summarizing format that was first implemented by Cisco Systems® primarily for accounting purposes. Network flow data (or network flow) is a generalization of NetFlow. Network flow collection differs from direct packet capture (such as with `tcpdump`) in that it builds a summary of communications between sources and destinations on a network. For NetFlow, this summary covers all traffic matching seven relevant keys: the source and destination IP addresses, the source and destination ports, the transport layer protocol, the type of service, and the router interface.

SiLK uses five of these attributes to constitute the *flow label*:

1. source IP address

2. destination IP address
3. source port
4. destination port
5. transport layer protocol

These attributes (also known as the *five-tuple*), together with the start time of each network flow, distinguish network flows from each other. The SiLK *repository* stores the accumulated flows from a network.

1.2.2 Structure of a Flow Record

A network flow often covers multiple packets that all match the fields of their common labels. A *flow record* thus provides the label and statistics on the packets covered by the network flow, including the number of packets covered by the flow, the total number of bytes, and the duration and timing of those packets (among other fields). A *flow file* is a series of flow records.

The fields in the flow record are listed in Table 1.1 (deprecated fields removed for clarity). Every field is identified by a name and number that can be used interchangeably. For example, the source IP address field of a flow record can be identified by either its field name (`sIP`) or its field number (`1`). Capitalization does not matter: `sIP` is equivalent to `sip` or `SIP`.

Because network flow is a summary of traffic, it does not contain packet payload data, which are expensive to retain on a large, busy network. Each network flow record created by SiLK is very small: it can be as little as 22 bytes (the exact size is determined by several configuration parameters). However, even at that tiny size, a sensor may collect many gigabytes of flow records daily on a busy network.

Some of the fields are actually stored in the record, such as start time and duration. Some fields are not actually stored; rather, they are derived either wholly from information in the stored fields or from a combination of fields stored in the record and external data. For example, end time is derived by adding the start time and the duration. Source country code is derived from the source IP address and a table that maps IP addresses to country codes.

1.2.3 Flow Generation and Collection

To understand how to use SiLK for analysis, it helps to have some knowledge of how network flow data are collected, stored, and managed. Understanding how the data are partitioned can produce faster queries by reducing the amount of data searched. In addition, by understanding how the sensors complement each other, it is possible to gather traffic data even when a specific sensor has failed.

Every day, SiLK may collect many gigabytes of network flow records from across the enterprise network. This section reviews the collection process and shows how data are stored as network flow records.

A network flow record is generated by sensors throughout the enterprise network. Usually, the majority of these sensors are routers. Specialized sensors such as `yaf`[2] can be employed when a data feed from a router is not available, such as on a home network or on an individual host. `yaf` can also be used to avoid artifacts in a router's implementation of network flow or to use non-device-specific network flow data formats such

[2]YAF (Yet Another Flowmeter). *CERT NetSA Security Suite website.* [Accessed May 14, 2020] https://tools.netsa.cert.org/yaf/

Field Number	Field Name	Description
1	sIP	Source IP address for flow
2	dIP	Destination IP address for flow
3	sPort	Source port for flow (or 0)
4	dPort	Destination port for flow (or 0)
5	protocol	Transport layer protocol number for flow
6	packets, pkts	Number of packets in flow
7	bytes	Number of bytes in flow (starting with IP header)
8	flags	Cumulative TCP flag fields of flow (or blank)
9	sTime	Start date and time of flow
10	duration	Duration of flow
11	eTime	End date and time of flow
12	sensor	Sensor that collected the flow
13	in	Ingress interface or VLAN on sensor (usually zero)
14	out	Egress interface or VLAN on sensor (usually zero)
15	nhIP	Next-hop IP address (usually zero)
16	sType	Type of source IP address (pmap required)
17	dType	Type of destination IP address (pmap required)
18	scc	Source country code (pmap required)
19	dcc	Destination country code (pmap required)
20	class	Class of sensor that collected flow
21	type	Type of flow for this sensor class
—	iType	ICMP type for ICMP and ICMPv6 flows (SiLK V3.8.1+)
—	iCode	ICMP code for ICMP and ICMPv6 flows (SiLK V3.8.1+)
25	icmpTypeCode	Both ICMP type and code values (before SiLK V3.8.1)
26	initialFlags	TCP flags in initial packet
27	sessionFlags	TCP flags in remaining packets
28	attributes	Termination conditions
29	application	Standard port for application that produced the flow

Table 1.1: Fields in a SiLK Network Flow record

as IPFIX[3]. It provides more control over network flow record generation and can convert packet data to network flow records via a script that automates this process.

A sensor generates network flow records by grouping together packets that are closely related in time and have a common flow label. "Closely related" is defined by the sensor and typically set to around 30 seconds. Figure 1.1 shows the generation of flows from packets. Case 1 in that figure diagrams flow record generation when all the packets for a flow are contiguous and uninterrupted. Case 2 diagrams flow record generation when several flows are collected in parallel. Case 3 diagrams flow record generation when timeout occurs, as discussed below.

Network flow is an approximation of traffic. Routers and other sensors make a guess when they decide which packets belong to a flow. These guesses are not perfect; there are several well-known phenomena in which a long-lived session will be split into multiple flow records:

[3]See https://tools.ietf.org/html/rfc7011 for definitions of the IPFIX information elements; see the IPFIX protocol description and https://www.iana.org/assignments/ipfix for their descriptions.

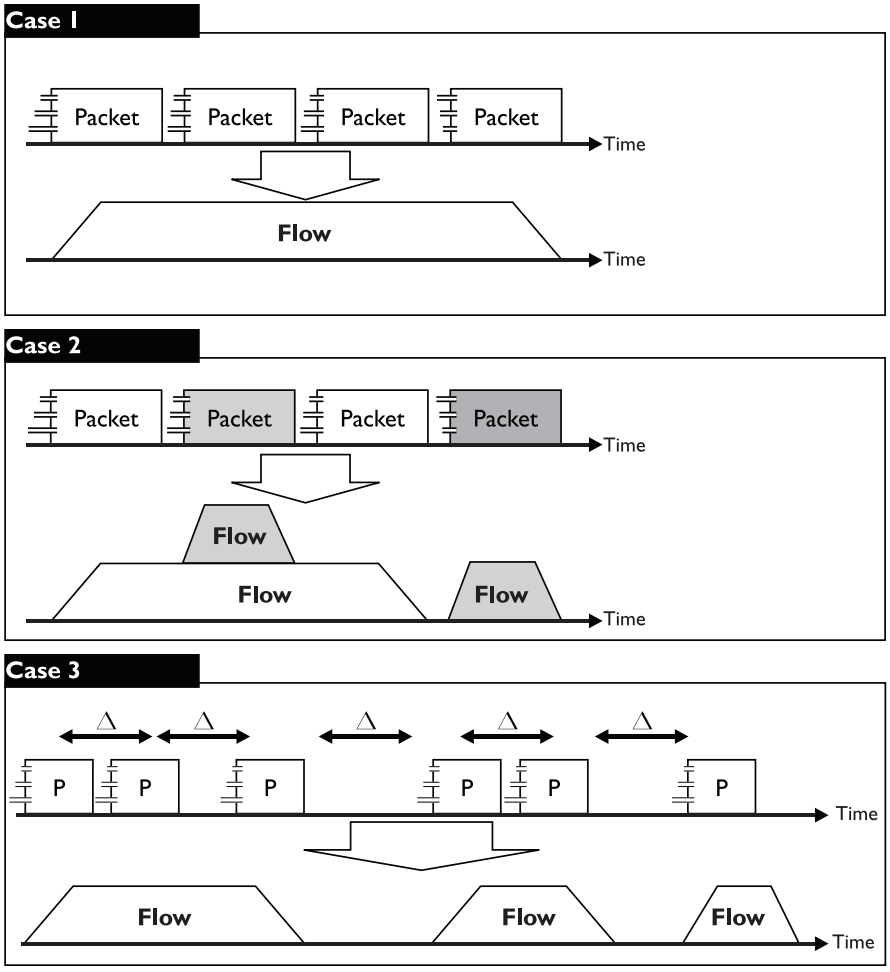

Figure 1.1: From Packets to Flows

1. *Active timeout* is the most common cause of a split network flow. Network flow records are purged from the sensor's memory and restarted after a configurable period of activity. As a result, all network flow records have an upper limit on their duration that depends on the local configuration. A typical value would be around 30 minutes.

2. *Cache flush* is a common cause of split network flows for router-collected network flow records. Network flows take up memory resources in the router, and the router regularly purges this cache of network flows for housekeeping purposes. The cache flush takes place approximately every 30 minutes as well. A plot of network flows over a long period of time shows many network flows terminate at regular 30-minute intervals, which is a result of the cache flush.

3. *Router exhaustion* also causes split network flows for router-collected flows. A router has limited processing and memory resources devoted to network flow. During periods of stress, the flow cache will fill and empty more frequently due to the number of network flows collected by the router.

Use of specialized flow sensors can avoid or minimize cache-flush and router-exhaustion issues. All of these

cases involve network flows that are long enough to be split. As we show later, the majority of network flows collected at the enterprise network border are small and short-lived.

1.2.4 Introduction to Flow Collection

An enterprise network comprises a variety of organizations and systems. The flow data to be handled by SiLK are first processed by the collection system, which receives flow records from the sensors and organizes them for later analysis. The collection system may collect data through a set of sensors that includes both routers and specialized sensors that are positioned throughout the enterprise network. After records are added to the flow repository by the collection system, analysis is performed using a custom set of software called the SiLK analysis tool suite.

The SiLK project is active, meaning that the system is continually improved. These improvements include new tools and revisions to existing collection and analysis software. See Appendix D for information on how to obtain the most up-to-date version of SiLK.

1.2.5 Where Network Flow Data Are Collected

While complex networks may segregate flow records based on where the records were collected (e.g., the network border, major points within the border, at other points), the generic implementation of the SiLK collection system defaults to collection only at the network border, as shown in Figure 1.2. The default implementation has only one class of sensors: *all*. Further segregation of the data is done by type of traffic.

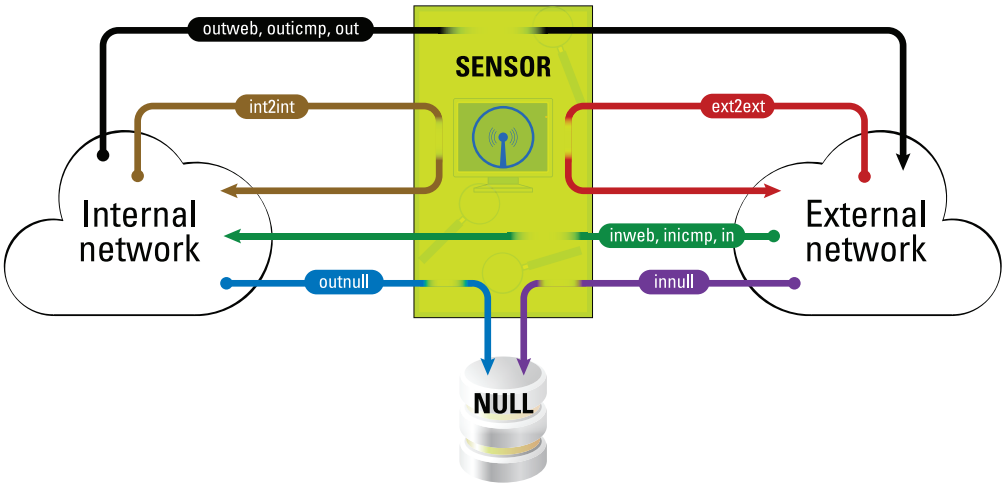

Figure 1.2: Default Traffic Types for Sensors

The SiLK tool `rwsiteinfo` can produce a list of sensors in use for a specific installation, reflecting its configuration. For more information on how to use this tool, see Section 2.2.1.

1.2.6 Types of Enterprise Network Traffic

In SiLK, the term *type* mostly refers to the direction of traffic, rather than a content-based characteristic. In the generic implementation (as shown in Figure 1.2), there are six basic types and five additional types. The basic types are

- `in` and `inweb`, which is traffic coming from the Internet service provider (ISP) to the enterprise network through the border router. Web traffic is separated from other traffic due to its volume, making many searches faster.
- `out` and `outweb`, which is traffic coming from the enterprise network to the ISP through the border router.
- `int2int`, which is traffic going both from and to the enterprise network, but which passes by the sensor.
- `ext2ext`, which is traffic going both from and to the ISP, but which passes by the sensor. (The presence of this type of traffic usually indicates a configuration problem either in the sensor or at the ISP.)

The additional SiLK types are

- `inicmp` and `outicmp`, which represent ICMP traffic entering or leaving the enterprise network. These types are operational only if SiLK was compiled with the option to support them.
- `innull` and `outnull`, which only can be found when the sensor is a router and not a dedicated sensor. They represent traffic from the upstream ISP or the enterprise network, respectively, that terminates at the router's IP address or is dropped by the router due to an access control list.
- `other`, which is assigned to traffic where one of the addresses (source or destination) is in neither the internal nor the external networks.
- The constructed type `all` selects all types of flows associated with a class of sensors.

These types are configurable. Configurations vary as to which types are in actual use (see the discussion below under Sensors: Class and Type).

1.2.7 The Collection System and Data Management

Data collection starts when a flow record is generated by one of the sensors: either a router or a dedicated sensor. Flow records are generated when a packet relevant to the flow is seen, but a flow is not *reported* until it is complete or flushed from the cache. Consequently, a flow can be seen some time after the start time of the first packet in the flow, depending on timeout configuration and on sensor caching, among other factors.

Packed flows are stored into files indicated by class, type, sensor, and the hour in which the flow started. So for traffic coming from the ISP through or past the sensor named SEN1 on March 1, 2018 for flows starting between 3:00 and 3:59:59.999 p.m. Coordinated Universal Time (UTC), a sample path to the file could be `/data/SEN1/in/2018/03/01/in-SEN1_20180301.15`.

1.2.8 How Network Flow Data Are Organized

The data repository is accessed via the SiLK tools, particularly the `rwfilter` command. An analyst uses `rwfilter` to choose the type of data to be viewed by specifying a set of selection parameters. This handbook discusses selection parameters in more detail in Section 2.2.2; this section briefly outlines how data are stored in the repository.

Dates

The SiLK repository stores data in hourly divisions, which are referred to in the form *yyyy/mm/dd*T*hh* in UTC. Thus, the hour beginning 11 a.m. on February 23, 2018 in Pittsburgh would be referred to as `2018/2/23T16` when compensating for the difference between UTC and Eastern Standard Time (EST)—five hours.

In general, data for a particular hour starts being recorded at that hour and will continue recording until some time after the end of the hour. Under ideal conditions, the last long-lived flows will be written to the file soon after they time out (e.g., if the active timeout period is 30 minutes, the last flows will be written out 30 minutes plus propagation time after the end of the hour). Under adverse network conditions, however, flows could accumulate on the sensor until they can be delivered. Under normal conditions, the file for 2018/3/7 20:00 UTC would have data starting at 3 p.m. in Pittsburgh and finish being updated after 4:30 p.m. in Pittsburgh.

Sensors: Class and Type

Data are divided by time and sensor. The class of a sensor is often associated with the sensor's role as a router: access layer, distribution layer, core (backbone) layer, or border (edge) router. The classes of sensors that are available are determined by the installation. By default, there is only one class—`all`—but based on analytical interest, other classes may be configured as needed. As shown in Figure 1.2, each class of sensor has several types of traffic associated with it: typically `in`, `inweb`, `out`, and `outweb`.

Data types are used for two reasons:

1. They group data together into common directions.
2. They split off major query classes.

As shown in Figure 1.2, most data types have a companion web type (i.e., `in` and `inweb`, `out` and `outweb`). Web traffic generally constitutes about 50% of the flows in any direction; by splitting the web traffic into a separate type, we reduce query time.

Most queries to repository data access one *class* of data at a time but access multiple *types* simultaneously.

1.3 The SiLK Tool Suite

The SiLK analysis suite consists of over 60 command-line UNIX tools (including flow collection tools) that rapidly process flow records or manipulate ancillary data. The tools can communicate with each other and with scripting tools via pipes (both unnamed and named) or via intermediate files.

Flow analysis is generally input/output bound (I/O bound)—the amount of time required to perform an analysis is proportional to the amount of data read from disk. A major goal of the SiLK tool suite is to

minimize that access time. Some SiLK tools perform functions analogous to common UNIX command-line tools and to higher level scripting languages such as Perl®. However, the SiLK tools process this data in non-text (binary) form and use data structures specifically optimized for analysis.

Consequently, most SiLK analysis consists of a sequence of operations using the SiLK tools. These operations typically start with an initial `rwfilter` call to retrieve data of interest and culminate in a final call to a text output tool like `rwstats` or `rwuniq` to summarize the data for presentation.

Keeping data in binary for as many steps as possible greatly improves efficiency of processing. This is because the structured binary records created by the SiLK tools are readily decomposed without parsing, their fields are compact, and the fields are already in a format that is ready for calculations, such as computing netmasks.

In some ways, it is appropriate to think of SiLK as an awareness toolkit. The flow-record repository provides large volumes of data, and the tool suite provides the capabilities needed to process these data. However, the actual insights come from analysts.

1.4 How to Use SiLK for Analysis

The SiLK tool suite provides a robust collection of tools to facilitate network traffic analysis tasks. It is designed to be very flexible in its support of analysis methods. Over time, different analysts have used a variety of approaches in their use of SiLK. This section discusses three approaches that have been useful in analyzing network flow records.

The chapters following this one expand on these approaches in more detail, focusing on the support that network flow analysis can provide to such analyses. Being aware of and practicing multiple approaches to analysis enables an analyst to gain insight into a wide variety of network traffic behaviors.

1.4.1 Single-path Analysis

The *single-path* approach is the most basic and most commonly-used approach to analyzing network behavior. It makes use of a single sequence of commands to produce the analytic results. In this approach, the analyst formulates an initial hypothesis, constructs a query to retrieve traffic of interest, produces a table, summary, or series to profile this traffic, and then interprets this profile either numerically or through a graph. Iteration can be used if needed (e.g., to refine the initial query), but may not be necessary for many simpler, more straightforward analyses.

This approach could be used for service identification, network device inventories, incident response, or usage studies. Chapter 2 provides an overview of single-path analysis, including the SiLK commands that are most commonly used with it. Chapter 3 describes example case studies of single-path analyses.

1.4.2 Multi-path Analysis

The *multi-path* approach uses a sequence of tools that frequently involve several alternatives, and often includes iterating over some steps. Although a multi-path approach can be done manually, it more often involves scripting to select alternatives based on categories of data and then iterate until the desired traffic is isolated or the desired summaries are produced. The alternatives are used as required for processing groups of records in differing ways to reach results that profile behavior of interest.

This approach could be used for examining traffic using several protocols, each following its own alternative set of characteristics, to accomplish the same goal. For example, there are multiple ways that malware can beacon to its command-and-control network. Each of those ways could be examined separately via a chain of SiLK commands, generating sets of results that contribute to overall awareness of beaconing.

Chapter 4 provides an overview of multi-path analysis, including the SiLK commands that are most commonly used with it. Chapter 5 describes an example case study of multi-path analysis.

1.4.3 Exploratory Analysis

We do not always know ahead of time what the scope of our analysis will be—or even what questions we should be asking! *Exploratory* analysis is an open-ended approach to formulating, scoping, and conducting a network analysis. It uses single-path and multi-path analyses as building blocks for investigating anomalous network traffic. These simpler types of analysis help us to formulate different scenarios, investigate alternative hypotheses, and explore multiple aspects of the data. Exploratory analysis is initially manual in nature, but can transition to scripted analysis for ease of repetition and for regularity of results.

This approach is used for complex or emerging phenomena, where multiple indicators need to be combined to gain understanding. An example of this approach to analysis would be a study of data exfiltration, which can be performed in a wide variety of ways. Each of those exfiltration methods could be profiled using a set of indicators, and the results of all such analyses combined to produce a composite understanding of traffic being passed to various groups of suspicious addresses.

Chapter 6 provides an overview of exploratory analysis, including advanced SiLK commands and concepts. Chapter 7 describes an example case study of exploratory analysis.

1.5 Workflow for SiLK Analysis

SiLK analyses share a common workflow, shown in Figure 1.3. While single-path, multi-path, and exploratory analysis may incorporate different steps in this workflow, all follow its general sequence.

1.5.1 Formulate

The *Formulate* step investigates the context of the event. Essentially, it involves collecting information to identify the unique attributes of the network, its operation, and the event. How large is the network? How is it structured? Where are network sensors located? When did the event occur? Is it associated with specific sensors, IP addresses, hosts, network spaces, ports, protocols, and so forth? Do any earlier analyses of the network offer insight? The information may be incomplete at this point, but it serves as a starting point for launching the analysis and establishing its scope. We can use it to formulate a hypothesis for the network's behavior. This hypothesis serves as the basis of our analysis. In more sophisticated exploratory analyses, we can formulate multiple scenarios and hypotheses for investigation and analysis.

Information gleaned from exploring the event's context helps us to establish which network behaviors should be included in (or excluded from) our analysis. We can use this information to construct a *query* to select and partition network flow records from the SiLK repository or a stored file. Queries typically incorporate information such as where the flow was collected, the date of data collection, and the flow direction. Within the SiLK community, query selection is commonly called a *data pull*.

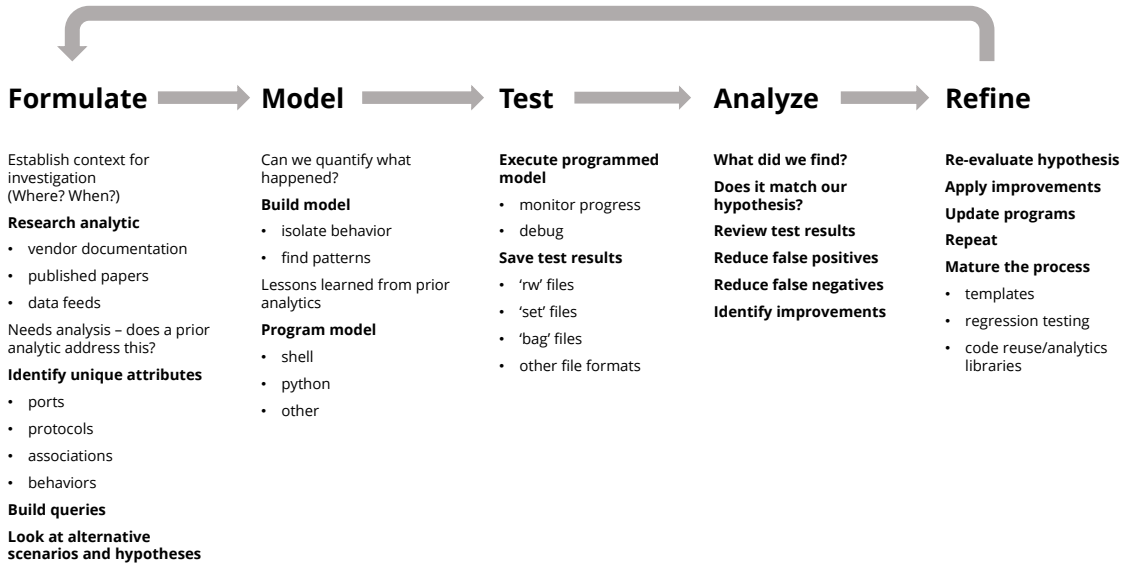

Figure 1.3: SiLK Analysis Workflow

Partitioning applies tests to selected flow records to separate (or partition) them into categories for further inspection and investigation. A default set of tests is provided with SiLK. It includes IP addresses, ports, protocols, time, and volumes. (If additional tests are needed for analyses, the SiLK tools can be extended via *plugins* to provide them.)

The combination of selection and partitioning (commonly referred to as *filtering*) is performed with the `rwfilter` command. Records that meet the filtering criteria are sent to *pass* destinations. Records that do not are sent to *fail* destinations. Both can be combined into *all* destinations. This provides flexibile options to either store query results in files or use pipes to send them to other commands for processing.

1.5.2 Model

The *Model* step summarizes data and investigates behaviors of interest. What is the network's behavior during normal operation? What happened during an event? What patterns and behaviors can we identify? Are they similar to those observed during other events? By examining the information gathered during the Formulate step, you can come up with a model of the event that perhaps explains what is going on.

SiLK provides a variety of tools for examining network flow data associated with an event. Each tool offers different views into the data that can be considered independently or in combination for analysis. For example, SiLK includes tools for generating time-series summaries of traffic (the `rwcount` command), computing summary statistics (the `rwstats` command), and summing up the values of flow attributes for user-defined time periods (the `rwuniq` command).

This step can be done manually. For analyses that are larger in scope, it can be automated by using shell or Python scripts.

1.5.3 Test

The *Test* step runs the model that you created—either manually or by executing shell or Python scripts. This gives you a chance to check the progress of the analysis.

SiLK includes commands for sorting flow records according to user-defined keys (the `rwsort` command), creating sets of unique IP addresses from flow records (`rwset` and its related commands), and creating groups of records by other criteria (`rwbag` and its related commands). These commands help you to organize output from the various SiLK commands and save it for further use.

1.5.4 Analyze

The *Analyze* step reviews the results of the previous steps. What do these results tell us about the event? What behaviors have been identified? What types of events are they associated with? What relationships can we identify between flows? Do our initial hypotheses still hold up? Can we find and eliminate false positives and false negatives?

This step involves examining and interpreting output from the analysis tools mentioned earlier. SiLK can also translate binary flow records into text for analysis with graphics packages, spreadsheets, and mathematical tools (the `rwcut` command).

1.5.5 Refine

The *Refine* step improves the analysis. Did we successfully explain the event? If not, what problems did we encounter? Did we properly understand the event's context? Did our query into the SiLK repository pull too much data? Do we need to dig deeper into the data during the modeling and testing steps? Should we take another look at the results to see if we missed or misinterpreted important patterns and behaviors?

The preceding steps in the workflow can be combined in an iterative pattern. For example, you may want to isolate flow records of interest from unrelated network traffic by making additional queries with the `rwfilter` command and repeating subsequent steps in the analysis. This narrows the data to focus on the time periods and behaviors of interest and eliminate unneeded flow records.

The workflow described in this section gives us the flexiblity to begin our data exploration with a general question, apply one or more analyses to the question, and complete the workflow with a repeatable analytic. This flexiblity does come with trade-offs, however. Queries typically increase proportionally with the time window and flow record attributes of an analysis. Therefore, a precise model of an analysis should be produced to minimize the query results.

1.6 Applying the SiLK Workflow

The SiLK workflow can be applied in different ways to meet the requirements of analysis groups. Groups that are primarily concerned with network operations will often focus on network monitoring or service and device validation. Incident response groups commonly focus on changes in network behavior that may be associated with an incident. Security improvement groups often focus on understanding problematic network behavior and changes that identify the impact of improvements. While the SiLK suite offers features that support all groups, the work required to use them will vary.

The extended examples and case studies in this handbook apply the SiLK workflow to perform common cybersecurity and network management operations. Most of them fall into three broad areas, each identified by an icon. When you see these icons, you'll know that the text that follows relates to these areas.

Situational awareness consists of understanding what is currently present on your network, knowing what is currently threatening it, and predicting what might be expected to happen in the future. The eye icon to the left identifies content related to situational awareness, including detailed examples of how to apply the SiLK analysis workflow to improve your awareness of what is happening on your network. Sections 2.3, 4.3, and 6.3 go into further detail abut using SiLK for situational awareness.

Incident Response is an organized approach to addressing and managing the aftermath of a security breach or cyberattack, also known as an IT incident, computer incident or security incident.[4] The magnifying glass icon to the left identifies content related to incident response, including detailed examples of how to apply the SiLK workflow to investigate and respond to security incidents.

Threat Hunting is proactively searching for malware or attackers that are lurking in a network — and may have been there for some time.[5] The binocular icon to the left identifies content related to threat hunting, including detailed examples of how to apply the SiLK workflow to find active and potential threats to your network.

1.7 Avoiding Cognitive Biases

Cognitive biases are systematic errors in thinking that affect the decisions and judgments that people make.[6] They grow out of limitations on how much information the human brain can process at a given time. Analysts need to take common cognitive biases into account when investigating network behavior. If these biases are not identified and countered, they can lead to analyses that are inaccurate or incomplete.

Common cognitive biases that can affect an analysis are listed below, along with strategies for avoiding them.

- **Anchoring bias** is where the analyst focuses too heavily on the initial profile or data within the profile. This bias results in a premature judgment of the threat.

  One way to fight anchoring bias is to assemble a balanced view across several datasets before making even tentative conclusions.

- **Selective perception bias** is the tendency to not notice parts of the profile that do not support assumptions about the threat. This bias results in an incomplete analysis of the threat.

  One way to fight this bias is to consider multiple possible outcomes before doing the analysis. Highlight data that supports each outcome before identifying the outcome with the strongest support.

- **Information bias** is where data is collected or analyzed improperly. This bias results in mistaken conclusions from analysis.

[4] Definition of incident response, *Techtarget Network website.* [Accessed May 5, 2020] https://searchsecurity.techtarget.com/definition/incident-response

[5] Byrne, Louse. A Beginner's Guide to Threat Hunting. *Security Intelligence website.* September 12, 2018. [Accessed May 5, 2020] https://securityintelligence.com/a-beginners-guide-to-threat-hunting/

[6] Cherry, Kendra. How Cognitive Biases Influence How You Think and Act. *VeryWellMind website.* [Accessed February, 18, 2020] https://www.verywellmind.com/what-is-a-cognitive-bias-2794963

One way to fight this bias is to double-check the collected data and use tested analytics to produce the profile.

- **Confirmation bias** is where new data is only interpreted as supporting an existing assessment of the threat. This bias prevents the analyst from considering alternative explanations and can result in an unsupported conclusion.

  One way to fight this bias is to consider multiple outcomes before analysis and highlight data that supports each one. (This is similar to the method for avoiding selective perception bias.)

- **Conservatism bias** is where an analyst fails to sufficiently revise the current interpretation of the threat when further information provides contrary evidence. This bias results in the analyst clinging to prior interpretations that are no longer supported by the current data, which leads to unreliable results.

  One way to fight this bias is to avoid even tentative conclusions until all current data is incorporated into the analysis. (This is similar to the method for avoiding anchoring bias.)

- **Recency bias** is when an analyst focuses too heavily on the most current results. This bias results in ongoing trends being discounted in favor of more recent variations, producing misleading results.

  One way to fight this bias is to examine both the longer-term trends and the recent variations before making conclusions about the threat.

1.8 Dataset for Single-path, Multi-path, and Exploratory Analysis Examples

The dataset used for the command examples and case studies of single-path, multi-path, and exploratory analysis in this document is the FCCX-15 dataset[7]. It originates from a June, 2015 Cyber Exercise conducted by the Software Engineering Institute at Carnegie Mellon University in a virtual environment.

The exercise network topology is shown in Figure 1.4 and is documented in the data download. It comprises a distributed enterprise for the period from June 2-16, 2015. Internet and transport layer protocols such as IPv4, TCP, UDP, and ICMP are well represented in the data. Link layer protocols such as IGMP and OSPF are also included; however, they are not as prevalent as the Internet and transport layer protocols.

[7] SiLK Reference Data. *CERT NetSA Security Suite website.* [Accessed May 5, 2020] https://tools.netsa.cert.org/silk/referencedata.html

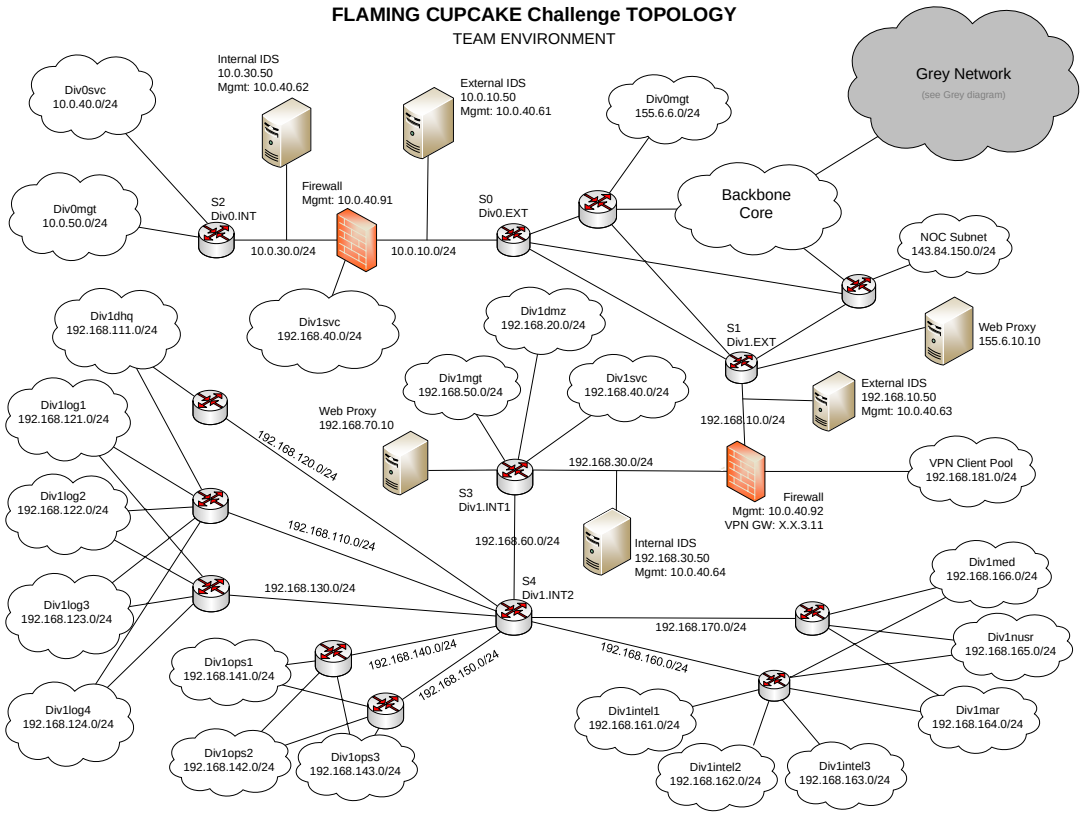

Figure 1.4: FCC Network Diagram

This page intentionally left blank.

Chapter 2

Basic Single-path Analysis with SiLK: Profiling and Reacting

This chapter introduces basic single-path analysis through application of the analytic development process with the SiLK tool suite. It discusses basic analysis of network flow data with SiLK, in addition to specific tools and how they can be combined to form a workflow.

Upon completion of this chapter you will be able to

- describe basic single-path analysis and how it maps to the analytic development process
- understand SiLK tools commonly used with basic single-path analysis
- describe SiLK IPsets and their application
- describe the single-path analysis workflow using network flow data
- describe how to apply the single-path analysis workflow to situational awareness

2.1 Single-path Analysis: Concepts

Single-path analysis is the approach of combining data with methods that do not require conditional steps, integration, or a great deal of refinement. In layman's terms, single-path analysis can be described as the 'start-to-finish' approach of combining one or more analytical steps to characterize network behavior. Its output may contain multiple attributes and characteristics; however, it results in information that normally does not need continued iteration. Figure 2.1 provides an overview of single-path analysis.

Single-path analysis typically incorporates the Formulate, Model, Test, and Analyze steps of the analysis workflow described in Section 1.5. The Refine step can also be included—for instance, to change the scope of a data pull from the SiLK repository—but is not always needed. The analysis begins by identifying the context of an analysis and formulating a hypothesis to explain the behavior under investigation. Event attributes such as hosts, networks, and time periods are used to identify, retrieve, and partition data for analysis. Attributes such as frequency, volume, and supporting network services provide additional behavioral context.

Single-path analysis then summarizes this data to produce sequences of event behavior. Data can be separated into logical groups such as successful and unsuccessful contacts, scanning, and misconfiguration. This enables analysts to produce known and unknown activity, trends, and differences for comprehensive analysis of a network's behavior.

Analysts also use single-path analyses as building blocks for broader, more complex analyses. See Chapter 4 and Chapter 6 for descriptions of analysis workflows that include single-path analyses as building blocks for more comprehensive investigations of network activity.

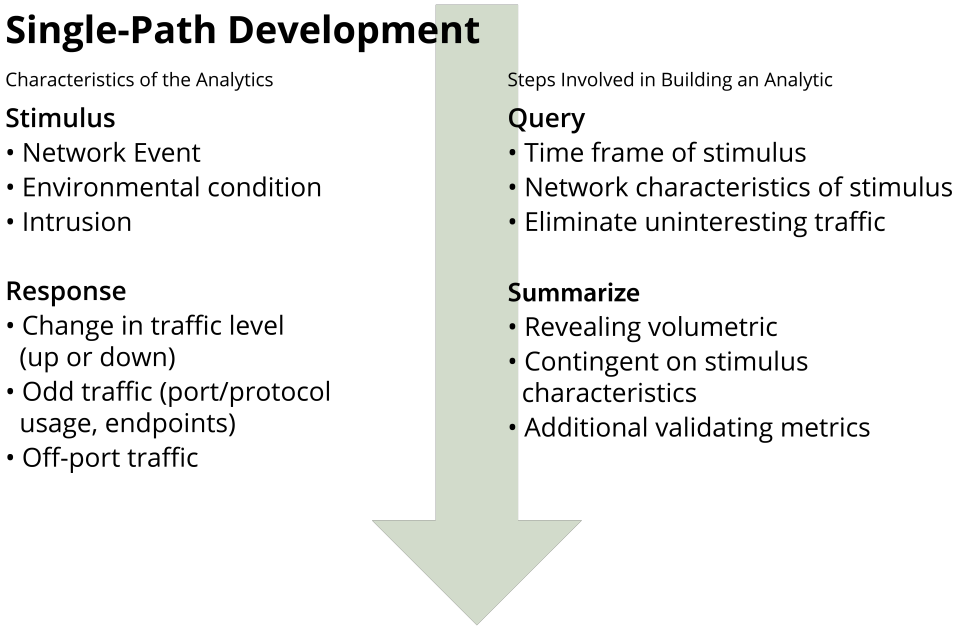

Figure 2.1: Single-Path Analysis

2.1.1 Scoping Queries of Network Flow Data

Be careful when defining the scope of an initial query into the SiLK repository or a file. The natural tendency is to make the partitioning criteria very inclusive, which has two drawbacks. It pulls over-large amounts of data, consuming storage and other computer resources. Overly-broad queries may match behaviors other than those of interest, which will complicate later steps in the analysis.

The preferred method for scoping queries is the opposite:

1. Make the partitioning criteria initially narrow, specific to the desired behaviors.
2. Once the traffic related to the behavior is retrieved, broaden the initial criteria to identify related network traffic.

Starting narrow and broadening the scope of the data query as the analysis proceeds will use computing resources more efficiently and facilitate clearer analysis by minimizing unwanted data.

2.1.2 Excluding Unwanted Network Traffic

Despite narrowing the initial query, unrelated traffic is commonly included in the intially-retrieved records. Analysis will often require isolating the desired activity from among the retrieved traffic. This may involve studying the traffic to identify unrelated behaviors and constructing further criteria to exclude them. It may also include eliminating traffic involving specific addresses (often by using the `rwset` tools to build IPsets), traffic that does not occur in the proper timeframe (often by using a further `rwfilter` call), or traffic that lacks specific protocol information associated with behavior of interest (again by using a further `rwfilter` call).

2.1.3 Example Single-Path Analysis

This chapter documents an example single-path analysis using the SiLK tool suite. It serves as an independent analysis, but could also depict the beginning of a multi-path (intermediate) or exploratory (advanced) analysis workflow. The example begins with identifying the context of an event by using `rwsiteinfo` to select relevant sensors and time periods for analysis. The time window is expanded beyond the event under analysis to select data to compare against the event period. `rwfilter` is then used to retrieve network flow records that apply to the defined sensor and period.

Traffic characteristics such as bytes, packets, and TCP flags options are then used with `rwfilter` on the retrieved data to select sequences of behaviors such as successful and unsuccessful contacts, scanning, and misconfigurations. The resulting network flow records are then displayed with tools such as `rwcut`, `rwuniq`, `rwcount`, and `rwstats`. These tools summarize and display network flow records using specified bins in order for analysts to verify and group data for traffic characterization and behavioral analysis. Top-N and bottom-N statistics, time series, event sequence, and record-by-record displays are a few examples depicted in this analysis.

Hosts that match specific characteristics or behaviors during an analysis are then saved to named SiLK IPsets. IPsets are data structures that represent an arbitrary collection of individual addresses, and are commonly named using a behavior, characteristic, or some other descriptive attribute. For example, `webservers.set` could be a IPset file of the source IP addresses obtained from querying network flow data for flows where the source IP address responded to a SYN scan on its port 80. These binary data structures enable analysts to use the SiLK tool suite to describe network traffic and save, display, or query the hosts that match those descriptions with tools such as `rwset`, `rwsetbuild`, `rwsetcat`, or `rwfilter`.

2.2 Single-path Analysis: Analytics

The commands, parameters, and examples described in this chapter serve as the building blocks for analyses with the SiLK tool suite.

2.2.1 Get a List of Sensors With `rwsiteinfo`

The first step in a basic, single-path network analysis of the dataset described in Section 1.8 is to find out which sensor recorded the data to be analyzed and narrow down the time period for our analysis. Since routing is relatively static, data from a specific IP address generally enters or leaves through the same sensor. You need to identify the sensor that covers the affected network and figure out when this sensor recorded network flow data.

Use the `rwsiteinfo` command to view this information for the sensors on your network. `rwsiteinfo` prints SiLK configuration file information for a site, its sensors, and the traffic they collect. The `--fields` parameter is required and specifies what information is displayed. Run `rwsiteinfo` twice to do the following:

1. List the names and descriptions of all the sensors on the network. This helps to locate the sensor that covers the affected network.
2. For the sensor of interest, list the types of SiLK traffic that it carries, the number of data files stored in the SiLK repository for each type of traffic, and the start and end times for storing network flow data in the repository. This identifies the direction and type of network traffic that the sensor recorded and the time period when it was actively storing data.

Example 2.1 shows the two `rwsiteinfo` commands and their output. The results of these two calls to `rwsiteinfo` will be used in Section 2.2.2 to build a query with the `rwfilter` command to select the network flow records for our analysis.

Determine Which Sensor Covers the Affected IP Addresses

To start, run the `rwsiteinfo` command to find the names and locations of the sensors in the network.

```
rwsiteinfo --fields=sensor,describe-sensor
```

The `--fields` parameters requests the following information:

- `sensor` displays the name of each sensor in plain text.
- `describe-sensor` displays the description of each sensor from the site configuration file (normally `silk.conf` in the root of the repository). A site's owner can specify information about the sensor configuration in this file. This gives you information (such as the sensor's location) that can help you to find which sensors recorded network traffic for the affected address block. (If the site's owner did not include this information in the site configuration file, nothing is displayed for this parameter.)

The output at the top of Example 2.1 lists the names and locations of the sensors. You need to find the sensor that covers the affected network. We are interested in traffic through the subnetwork `Div1Ext`. The sensor `S1` is associated with this subnetwork, which we will examine more closely.

Find Traffic Types and Repository Storage Times

Once you have found the sensor of interest (`S1`), you can find out what kinds of traffic the sensor carries and when it wrote data to the SiLK repository.

```
rwsiteinfo --sensor=S1 --fields=type,repo-file-count,repo-start-date,repo-end-date
```

- `--sensor` specifies which sensor to examine. In this example, it is the name of the sensor identified via the first `rwsiteinfo` command (`S1`).
- `--fields` displays the following information in table format for sensor `S1`:

```
<1>$ rwsiteinfo --fields=sensor,describe-sensor
Sensor|Sensor-Description|
    S0|           Div0Ext|
    S1|           Div1Ext|
    S2|           Div0Int|
    S3|          Div1Int1|
    S4|          Div1Int2|
    S5|          Div1log1|
    S6|          Div1log2|
    S7|          Div1log3|
    S8|          Div1log4|
    S9|          Div1ops1|
   S10|          Div1ops2|
   S11|          Div1ops3|
   S12|           Div1svc|
   S13|           Div1dhq|
   S14|           Div1dmz|
   S15|           Div1mar|
   S16|           Div1med|
   S17|          Div1nusr|
   S18|           Div1mgt|
   S19|        Div1intel1|
   S20|        Div1intel2|
   S21|        Div1intel3|
<2>$ rwsiteinfo --sensor=S1 \
   --fields=type,repo-file-count,repo-start-date,repo-end-date
   Type|File-Count|         Start-Date|           End-Date|
     in|       441|2015/06/02T13:00:00|2015/06/18T18:00:00|
    out|       512|2015/06/02T13:00:00|2015/06/18T18:00:00|
  inweb|       328|2015/06/02T13:00:00|2015/06/18T18:00:00|
 outweb|       446|2015/06/02T13:00:00|2015/06/18T18:00:00|
 innull|         0|                   |                   |
outnull|         0|                   |                   |
int2int|       511|2015/06/02T13:00:00|2015/06/18T18:00:00|
ext2ext|       204|2015/06/02T13:00:00|2015/06/18T18:00:00|
 inicmp|         0|                   |                   |
outicmp|         0|                   |                   |
  other|         0|                   |                   |
```

Example 2.1: Using `rwsiteinfo` to List Sensors, Display Traffic Types, and Show Repository Information

`type`—the types of enterprise network traffic that are associated with `S1`. This tells you the direction and origin of the network traffic it carries. It is also useful for splitting off data of interest (for instance, separating inbound Web traffic from other inbound traffic), which can speed up SiLK queries. (To learn more about the basic SiLK network types, see Sections 1.2.6 and 1.2.8.)

`repo-file-count`—the number of files that `S1` stored in the SiLK repository for each type of network traffic. Each file represents one hour of recorded data.

`repo-start-date`—the time and date of the oldest file that `S1` stored in the SiLK repository.

`repo-end-date`—the time and date of the most recent file that `S1` stored in the SiLK repository.

The output at the bottom of Example 2.1 lists the different types of network traffic carried by `S1`. The bulk of this traffic was recorded from `2015/06/02T13:00:00` through `2015/06/18T18:00:00`. In the next step of our analysis, we will therefore retrieve network flow records from `S1` within this time period.

The output from Example 2.1 can also tell us whether `S1` recorded enough data to support a meaningful network analysis. The repository contains 441 files of inbound traffic from the ISP to the network (`in`), representing 441 hours of recorded inbound traffic to the IP addresses covered by `S1`. Similarly, the repository contains 512 hours of outbound traffic from these IP addresses to the ISP (`out`), 328 hours of inbound Web traffic (`inweb`), and 446 hours of outbound web traffic (`outweb`). This is sufficient for our analysis.

This example shows the process described in Section 1.5. The formulation is to find which sensors recorded the data. The model used the static nature of the data and applied `rwsiteinfo`. Example 2.1 provided the test, producing data that was analyzed to yield the sensor. The refinement was to use further parameters with `rwsiteinfo` to determine traffic types and date ranges.

Help with `rwsiteinfo`

For a full list of `rwsiteinfo` options, type `rwsiteinfo --help`.

For a detailed description of the `rwsiteinfo` command, type `man rwsiteinfo` at the command prompt.

Other Useful `rwsiteinfo` Options

Keep the following in mind when using this command:

- You must always specify parameters with `rwsiteinfo`; there is no default output.
- Enter `rwsiteinfo --fields` options in the order that you would like them to be displayed. For instance, to view the sensor description before the sensor name, specify `--fields=describe-sensor,sensor`
- To find the classes and types supported by an installation, run `rwsiteinfo --fields=class,type,mark-defaults`. This produces three columns labeled `Class`, `Type`, and `Defaults`. The Defaults column shows plus signs (`+`) for all the types in the default class and asterisks (`*`) for the default types in each class.
- The `rwsiteinfo` command supports optional parameters to control the formatting of its output (disable column spaces, change separation character, disable column headers, change field separators). It can also limit output to specific network types of interest.

2.2.2 Choose Flow Records With `rwfilter`

A key step in performing a network analysis is to find and retrieve network flow records associated with the event from the SiLK repository. Use the `rwfilter` command to pull network flow records that were recorded by the sensor of interest (`S1`) during the time period of interest. These records will be used in subsequent steps in our analysis.

During this step in the analysis, `rwfilter` will be used to save network flow records of interest to a file. Later, we'll use `rwfilter` in conjunction with other SiLK commands to partition and explore this data.

About the `rwfilter` command

`rwfilter` is the most commonly used SiLK command and serves as the cornerstone for building a network analysis. It selects records from the SiLK repository, then directs the output to either files or other SiLK commands. Alternatively, `rwfilter` can select records from a pipe or file in a working directory (for instance, the output of a prior `rwfilter` command). It can optionally compute basic statistics about the flow records it reads from the repository or a file. `rwfilter` can be used on its own or in conjunction with other SiLK analysis tools, including additional invocations of `rwfilter`.

The following is a high-level view of the `rwfilter` command and its options:

`rwfilter` {*selection* | *input*} *partition output*

Specify input to `rwfilter` by using either selection or input parameters.

- *Selection* parameters read (or pull) network flow records of interest that were recorded by sensors and stored in the SiLK flow repository. They specify the attributes of records to be read from the repository, such as the sensor that recorded the data, the type of network data, the start and end dates for retrieving data, and the location of the repository.

- *Input* parameters read network flow records from pipes and/or named files in working directories containing records previously extracted from the repository or created by other means. They can be filenames (e.g., `infile.rw`) or pipe names (e.g., `stdin` or `/tmp/my.fifo`) to specify locations from which to read records. As many names as desired may be given, with both files and pipes used in the same command.

In this step of our network analysis, we will use `rwfilter`'s selection parameters to retrieve records from the SiLK repository. In future steps, we will use `rwfilter`'s input parameters to read flow records from a file or pipe.

Partitioning parameters create the "filter" part of `rwfilter`. These parameters specify which records pass the filter and which fail. This enables you to find and isolate network flow records that match the partitioning criteria you specify. `rwfilter` offers a variety of filtering parameters for specifying the criteria for pass/fail filtering, including time period, value ranges for packets and bytes, IP address, protocol, source and destination ports, and more.

Hint 2.1: Minimum Partitioning Parameters for `rwfilter`

An analysis will involve at least one call to `rwfilter` unless you are looking at records saved from a previous analysis. Each `rwfilter` call must include at least one partitioning parameter unless `--all-destination` is specified as an output parameter. Note that the partitioning parameter does not have to filter anything; it just needs to be present. The partitioning parameter `--protocol=0-` is often used in this situation since it will not filter any records.

In this step of our network analysis, we will specify just one partitioning parameter: the IP address that is associated with the event. In future steps, we will specify additional partitioning parameters to identify records of interest and isolate them for further exploration with other SiLK commands.

Output parameters specify which group of records is returned from the call to `rwfilter`: those that "pass" the filter, those that "fail" the filter, both, or neither. These records can be written to pipes and/or named files in a working directory via the output parameters. (This also applies to statistics computed with `rwfilter`.) Each call to `rwfilter` must have at least one output parameter.

In this step of our network analysis, we will use output parameters to specify the name of the file where records are stored. In future steps, we will use pipes to direct `rwfilter` output to other SiLK commands for further investigation and processing.

Hint 2.2: `rwfilter` File Naming Conventions

The examples in this guide store flow records in either `*.rw` or `*.raw` files (e.g., `flows.rw`, `external.raw`). Both are commonly-accepted file extensions for `rwfilter` output.

Figure 2.2 shows how the `rwfilter` parameters interact.

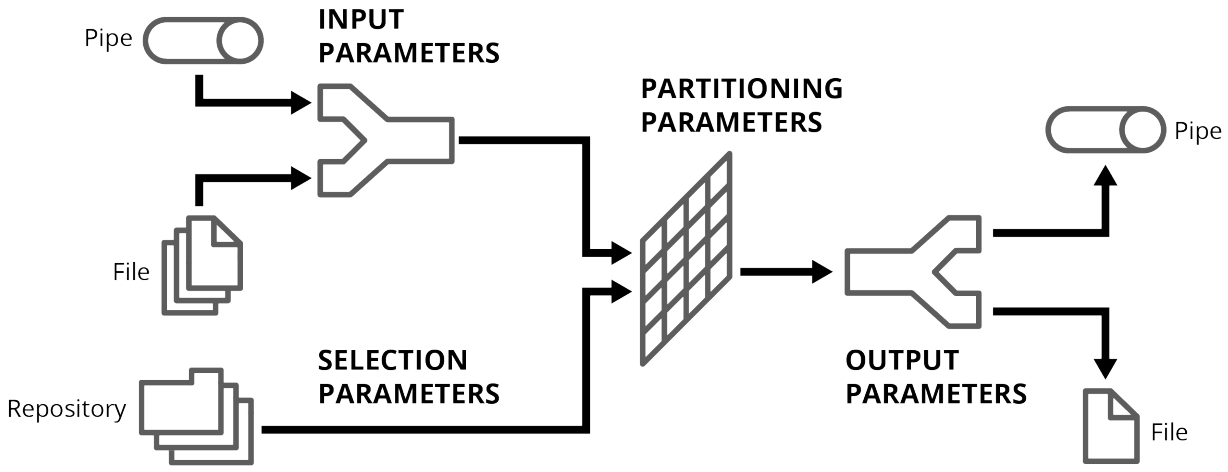

Figure 2.2: `rwfilter` Parameter Relationships

Retrieving network flow records and saving them to a file

Our sample single-path analysis pulls network flow records from the SiLK repository with `rwfilter`, saves them to a binary file, then examines the data in the file. This is often much faster and more efficient than pulling fresh data from the repository at every step in the analysis. `rwfilter` queries into large repositories can take a long time to run—especially if you are investigating activity over an extended period of time. To look at another group of records from the repository (for instance, from a different sensor or time period), simply run `rwfilter` again to retrieve the desired records and create additional files for analysis.

Use the `rwfilter` command as follows to pull network flow records associated with the sensor, time period, and IP address of interest to our analysis (demonstrated in Example 2.2):

```
rwfilter --start=2015/06/17T14 --end=2015/06/17T14 --sensor=S1 --type=all
--any-address=192.168.70.10 --pass=flows.rw
```

- `--start` and `--end` specify the time period for retrieving records from the SiLK repository (which was found in Section 2.2.1). Times can be expressed in the form *YYYY/MM/DD*T*hh*:*mm*:*ss.mmm*. While some `rwfilter` parameters use the full format, `--start` and `--end` use an abbreviated time form, specifying time to the day (`2015/06/17`) or to the hour `2015/06/17T14`), but no finer resolution than the hour. In this example, one hour of traffic is pulled from the repository, starting at UTC 14:00:00.000 and ending at UTC 14:59:59.999 on June 17, 2015.

Hint 2.3: SiLK Parameter Abbreviations

The full parameter names are `--start-date` and `--end-date`. SiLK will recognize a parameter as long as you specify enough of its name to uniquely identify it.

- `--sensor` specifies which sensor's records to retrieve (sensor `S1`, which was identified in Section 2.2.1)
- `--type` specifies the types of SiLK network traffic to retrieve. We will pull records for `all` network traffic.
- `--any-address` sets up a simple pass/fail filter for partitioning the selected network flow records. We are interested in traffic associated with the IP address 192.168.70.10. Records that match the specified IP address pass the filter; records that do not, fail it.
- `--pass` specifies the destination of the selected records that pass the filter. In this case, they are stored in the local disk file `flows.rw`.

Hint 2.4: `rwfilter` Output File Performance

Be aware that saving `rwfilter` output to a network disk file can slow down this command considerably. The speed at which records are written to the file is limited by the speed of the network. Saving to a local file is faster. (To learn more about strategies for speeding up `rwfilter` performance, see Chapter 9.)

The resulting binary file, `flows.rw`, contains network flow records from the time period, sensor, traffic types, and IP address of interest. The `ls` command in Example 2.2 shows that this file has content after the `rwfilter` command. In other words, the records in this file are a snapshot of the event that we will be investigating over the course of our network analysis.

```
<1>$ rwfilter --start=2015/06/17T14 --end=2015/06/17T14 \
   --sensor=S1 --type=all --any-address=192.168.70.10 \
   --pass=flows.rw
<2>$ ls -l flows.rw
-rw-r--r--. 1 analyst analyst 365935 Nov  3 14:48 flows.rw
```

Example 2.2: Using `rwfilter` to Retrieve Network Flow Records From The SiLK Repository

Help with `rwfilter`

Type `rwfilter --help` for a full list of parameters.

Type `man rwfilter` for a detailed description of the `rwfilter` command and its parameters.

Other Useful `rwfilter` Options

Keep the following in mind when using `rwfilter`:

- Some selection parameters can be used as partitioning parameters when `rwfilter` is pulling network flow records from a file or pipe. The `--sensor`, `--type`, `--class`, and `--flowtype` parameters can perform double duty for selecting and partitioning records.
- When specifying selection parameters, experienced analysts include a `--start-date` to avoid having `rwfilter` implicitly pull all records from the current day, potentially leading to inconsistent results.
- `rwfilter` partitioning parameters give analysts great flexibility in describing which flow records pass or fail the filter. Figuring out how to partition data to filter out unwanted records can be the most difficult part of using this command.
- Narrowing the selection of files from the repository always improves the performance of a query. On the other hand, increasing the specificity of partitioning options could improve or diminish performance. Increasing the number of partitioning parameters means more processing must be performed on each flow record. Most partitioning options involve minimal processing, but some involve considerable processing.

  Generally, processing partitioning options is much less of a concern than the number of output operations, especially disk operations, and most especially network disk operations. Choosing to output both the "pass" and "fail" sets of records will involve more output operations than choosing only one set.
- The parameter `--print-filenames` lists, on the standard error file, the name of each file as `rwfilter` opens it for reading. This provides assurance that the expected files were read and indicates the command's progress. (This is especially useful when many files are used as data sources and the command will take a long time to complete.)

- `rwfilter` can take multiple files and pipes as input. If the number of files exceeds what is convenient to put in the command line, use the `--xargs` parameter. It specifies the name of a file containing filenames from which to read flow records. This parameter also is used when another UNIX process is generating the list of input files, as in

```
find . -name '*.rw' | --xargs=stdin ...
```

2.2.3 View Flow Records With `rwcut`

Translating network flow records from binary format into human-readable text is a helpful part of a network analysis. Use the `rwcut` command to translate the binary network flow records selected via the `rwfilter` command as tables of ASCII text.

SiLK uses binary data to speed up queries, file manipulation, and other operations. However, these data cannot be read using any of the standard text-processing UNIX tools. `rwcut` reads SiLK flow records and translates this binary data into pipe-delimited (`|`) text output. You can then view the data directly in a terminal window or read it into a text-processing, graphing, or mathematical analysis tool.

Hint 2.5: Keep Data in Binary Format Where Possible

Keep data in binary format (i.e., `*.rw` files) for as long as possible while performing an analysis. Binary SiLK network flow records are more compact and offer faster performance than the ASCII representation of these records. Use `rwcut` to inspect records or export data to other tools for further analysis. See Chapter 9 for examples of text vs. binary processing.

`rwcut` can be invoked in two ways: by reading a file or by connecting it with another SiLK tool (such as `rwfilter` or `rwsort`) via a pipe. When reading a file, specify the file name in the command line. The `--fields` parameter selects, reorders, and formats SiLK data fields as text and separates them in different ways.

Displaying Flow Records

As part of the network analysis, in Example 2.3 we will use `rwcut` to take a closer look at a set of flow records from the file `flows.rw` (produced in Example 2.2).

```
rwcut --fields=sip,dip,sport,dport,protocol,stime --num-recs=10 flows.rw
```

- The `--fields` parameter specifies which fields in a SiLK record are shown. Field names are case-insensitive. This example displays the following fields:

  `sip`—source IP address for the flow

  `dip` —destination IP address for the flow

  `sport`—source port for the flow

  `dport`—destination port for the flow

  `protocol`—transport-layer protocol for the flow

`stime`—start time of the flow, formatted as *YYYY/MM/DD*T*hh:mm:ss.mmm*

- The `--num-recs` parameter determines how many records `rwcut` displays. In this example, up to ten records are shown (regardless of how many records are actually in the file). If the file were to contain no records, `rwcut` only displays the column heading for each field.
- `flows.rw` is the name of the file containing SiLK network flow records.

```
<1>$ rwcut --fields=sip,dip,sport,dport,protocol,stime \
   --num-recs=10 --ipv6-policy=ignore flows.rw
             sIP|             dIP|sPort|dPort|pro|                  sTime|
     10.0.40.83|  192.168.70.10|53981| 8082|  6|2015/06/17T14:00:02.631|
     10.0.40.20|  192.168.70.10|   53|58887| 17|2015/06/17T14:00:04.619|
     10.0.40.20|  192.168.70.10|   53|55004| 17|2015/06/17T14:00:04.621|
     10.0.40.83|  192.168.70.10|53982| 8082|  6|2015/06/17T14:00:12.673|
     10.0.40.20|  192.168.70.10|   53|64408| 17|2015/06/17T14:00:14.685|
     10.0.40.20|  192.168.70.10|   53|57734| 17|2015/06/17T14:00:14.689|
     10.0.40.83|  192.168.70.10|53983| 8082|  6|2015/06/17T14:00:22.709|
     10.0.40.20|  192.168.70.10|   53|63770| 17|2015/06/17T14:00:24.753|
     10.0.40.20|  192.168.70.10|   53|53374| 17|2015/06/17T14:00:24.755|
     10.0.40.83|  192.168.70.10|53984| 8082|  6|2015/06/17T14:00:32.741|
```

Example 2.3: `rwcut` for Displaying the Contents of Ten Flow Records

The six fields specified with the `rwcut` command are displayed in the order in which they are listed. Each field is in a separate column with its own header. The source IP addresses (`sip`) for each record vary; two addresses are shown in this example. The destination IP address (`dip`) is the same for all of these records since we only pulled records associated with that IP address. The output in Example 2.3 shows that the host of interest (`192.168.70.10`) is receiving DNS responses (protocol 17 (UDP), source port 53 (DNS)) and service requests to TCP (protocol 6) destination port 8082 (associated with a file server).

Hint 2.6: Format IP Addresses with SiLK Environment Variables

Your output may contain additional spaces in the IP address field. The environment variable `SILK_IPV6_POLICY=ignore` ignores any flow record marked as IPv6, regardless of the IP addresses it contains. Only records marked as IPv4 will be printed. Setting this environment variable has the same effect as invoking `rwcut` with the `--ipv6-policy=ignore` parameter.

Help with `rwcut`

Type `rwcut --help` for a full list of parameters.

Type `man rwcut` for a detailed description of the `rwcut` command and its parameters.

Other Useful `rwcut` Options

Keep the following in mind while using the `rwcut` command:

- The `--fields` parameter selects which network flow record fields appear in `rwcut` output. Each field is associated with a number as well as a name. Table 1.1 lists the field numbers and their corresponding field names. Numbers can be specified individually or as ranges. Field names and numbers can be combined and can be listed in arbitrary order. For instance, `--fields=1-4,9,protocol` produces the same output as `--fields=sip,dip,sport,dport,stime,protocol`
- The `--fields` parameter also specifies the order in which fields are shown in output. Fields can be displayed in any order. Example 2.4 displays the output fields in this order: source IP address, source port, start time, destination IP address. This is useful to emphasize trends. By this rearrangement, we can more easily see that the TCP traffic is using sequential ephemeral source ports, and being sent at 10 second separations. Both of these features indicate that this is manufactured traffic.

```
<1>$ rwfilter flows.rw --protocol=6 --max-pass-records=4 \
   --pass=stdout | rwcut --fields=1,3,sTime,2
            sIP|sPort|                  sTime|            dIP|
     10.0.40.83|53981|2015/06/17T14:00:02.631|  192.168.70.10|
     10.0.40.83|53982|2015/06/17T14:00:12.673|  192.168.70.10|
     10.0.40.83|53983|2015/06/17T14:00:22.709|  192.168.70.10|
     10.0.40.83|53984|2015/06/17T14:00:32.741|  192.168.70.10|
```

Example 2.4: `rwcut --fields` to Rearrange Output

- If `--fields` is not specified, `rwcut` prints the source and destination IP address, source and destination port, protocol, packet count, byte count, TCP flags, start time, duration, end time, and the sensor name.
- The `--delimited=`*C* parameter changes the separator from a pipe (`|`) to any other single character, where *C* is the delimiting character. It also removes spacing between fields. This is particularly useful with `--delimited=','` which produces comma-separated-value (CSV) output for easy import into spreadsheet programs and other tools that accept CSV files. `--delimited` is the equivalent of specifying `--no-columns --no-final-delimiter --column-sep=`*C*.
- When output is sent to a terminal, `rwcut` (and other text-outputting tools) automatically invoke the command listed in the user's `PAGER` environment variable to paginate the output. The command given in the `SILK_PAGER` environment variable will override the user's `PAGER` environment. If `SILK_PAGER` contains the empty string, no paging will be performed. The paging program can be specified for an individual command invocation by using its `--pager` parameter.

2.2.4 Viewing File Information with `rwfileinfo`

Analyses using the SiLK tool suite can become quite complex, with several intermediate files created while isolating the behavior of interest. The `rwfileinfo` displays a variety of characteristics for each file format produced by the SiLK tool suite, which helps you to manage these files. Use this command (demonstrated in Example 2.5) to find out more information about the file `flows.rw`, which contains the SiLK records associated with the IP address of interest.

For most analysts, the three most important file characteristics are the number of records in the file, the size of the file, and the SiLK commands that produced the file. Enter the following `rwfileinfo` command to view this information for `flows.rw`:

```
rwfileinfo --fields=count-records,file-size,command-lines flows.rw
```

- `--fields` specifies which SiLK file characteristics are displayed.

  `count-records`—the total number of network flow records in a flow record file.

  `file-size`—the size of the file in bytes.

  `command-lines`—the commands used to generate the file. This can be very helpful when performing an analysis that involves many steps and repeated applications of commands such as `rwfilter`.

- `flows.rw` is the name of the file containing SiLK network flow records.

Output is shown in Example 2.5. `flows.rw` contains 21,864 network flow records; its size is 365,935 bytes. This gives us an idea of how much network traffic is stored there. The SiLK command that generated the file is the `rwfilter` command described in Section 2.2.2.

```
<1>$ rwfileinfo --fields=count-records,file-size,command-lines \
   flows.rw
flows.rw:
  count-records        21864
  file-size            365935
  command-lines
                   1  rwfilter --start=2015/06/17T14 --end=2015/06/17T14 \
  --sensor=S1 --type=all --any-address=192.168.70.10 --pass=flows.rw
```

Example 2.5: `rwfileinfo` Displays Flow Record File Characteristics

Help with `rwfileinfo`

Type `rwfileinfo --help` for a full list of parameters.

Type `man rwfileinfo` for a detailed description of the `rwfileinfo` command and its parameters.

Other Useful `rwfileinfo` Options

Keep the following in mind while using the `rwfileinfo` command:

- While `rwfileinfo` is generally associated with flow record files, it can also show information on sets, bags, and prefix maps (or pmaps). For more information, see Section 2.2.8, Section 4.2.4, and Section 6.2.7, respectively.
- Be sure to use the `--fields` parameter to choose which network flow record fields are displayed. If no fields are specified, `rwfileinfo` defaults to displaying a dozen fields—many of which are of no use to analysts.
- For flow record files, the record count is the number of flow records in the file. For files with variable-length records (indicated by a `record-length` of one) the field does *not* reflect the number of records; instead it is the uncompressed size (in bytes) of the data section. Notably, `count-records` does not reflect the number of addresses in an IPset file.

2.2.5 Profile Flows With `rwuniq` and `rwstats`

The next step in our network analysis is to investigate network flows to and from the IP address of interest. We will determine the most common protocols associated with these flows and find flows with low, medium, and high byte counts.

Two SiLK commands can perform these tasks.

- `rwuniq` is a general-purpose counting tool. It reads binary SiLK flow records from a file (or standard input) and counts the records, bytes, and packets for any combination of fields. `rwuniq` also groups (or *bins*) the records by a time interval specified by the analyst.

  In our example, this command is used to identify the hour-long time bins containing flows with low, medium, and high byte counts.

- `rwstats` provides a collection of statistical summary and counting facilities that organize and rank traffic according to various attributes. It reads binary SiLK flow records from a file (or standard input) and then either computes summary statistics or groups them according to a key composed of user-specified flow fields, such as address, port, protocol, or detected application.

  `rwstats` can also bin the records by a user-specified start-time or end-time interval. For each bin, it sums up the values of volumetric attributes, such as bytes, packets, flow records, or unique values of flow attributes that are not part of the key, and sorts the bins by volumetric attribute. `rwstats` can find the N highest or N lowest bins. It also sums up the attribute values across all of the records it counts and displays the count for each bin as a percentage of the total.

  In our example, the `rwstats` command is used to identify the most commonly-used protocols associated with traffic to and from the IP address of interest.

`rwuniq` and `rwstats` overlap in their functions. Both assign flows to time bins by a key that is set by the analyst. The key represents combinations of flow field values for the records collected, not the number of records in each bin. For each value of a key (specified by the `--fields` parameter), a bin contains counts of flows or some other measure (specified with the `--values` parameter). `rwuniq` displays one row of output for every bin that falls within a threshold specified by the analyst. `rwstats` displays one row of output for each bin in the top N or bottom N of the total count, and computes the percentages of each data types. For a more detailed discussion of when to use each command, see Comparing `rwstats` to `rwuniq` (later in this section).

Finding Low, Medium, and High-Byte Flows with `rwuniq`

First, use the `rwuniq` command to profile flows by byte count. It can find out how many network flow records within an hour-long period have a low byte count (between zero and 300 bytes), a medium byte count (between 300 and 100,000 bytes), or a high byte count (more than 100,000 bytes). This gives you an estimate of the volume of network activity associated with the IP address of interest.

To perform this analysis, use the `rwuniq` command in conjunction with the `rwfilter` command.

1. Run `rwfilter` on the `flows.rw` file. This file contains all traffic to and from the IP address of interest during the time period of interest; it was extracted in Section 2.2.2. Running `rwfilter` on it a second time pulls all of the records in the file with the specified byte ranges.
2. Use the Unix pipe (|) command to direct the resulting output to the `rwuniq` command. This command counts the number of records with each range of bytes and directs the output to a file.

```
<1> $  rwfilter flows.rw --bytes=0-300 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type  --values=records --sort-output \
> low-byte.txt
<2> $  rwfilter flows.rw --bytes=300-100000 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type  --values=records --sort-output \
> medium-byte.txt
<3> $  rwfilter flows.rw --bytes=100000- --pass=stdout --values=records \
| rwuniq --bin-time=3600 --fields=stime,type  --sort-output \
> high-byte.txt
```

Parameters for the `rwfilter` command include the following:

- `flows.rw` contains the network flow records of interest.
- `--bytes` specifies the range of byte counts for selecting records. Ranges are specified using a dash (e.g., `0-300` selects all flows with byte counts between zero and 300). To specify an open-ended range, do not include an upper bound on the range (e.g., `100000-` selects all flows with byte counts equal to or greater than 100000).
- `--pass=stdout` sends all records that pass the filter to standard output.

Parameters for the `rwuniq` command include the following:

- `flows.rw` is the name of the file containing SiLK network flow records.
- `--bin-time=3600` defines a time bin that is one hour (3600 seconds) long.
- `--fields=stime,type` specifies the fields to use as keys for counting network flows. This parameter is required. We are looking at the values for `stime` (start time for the flow) and `type` (network flow type).
- `--values=records` counts the number of records that passed the `rwfilter` command.
- `--sort-output` sorts the output of the `rwuniq` command in numerical order according to the value (or values) of the key specified via the `--fields` parameter.
- The shell command `>` directs the output of `rwuniq` into the file `low-byte.txt`.

Hint 2.7: Use Unix Pipes To Improve SiLK Performance

We could have saved the `rwfilter` output to a file and run `rwuniq` on that file instead of using the UNIX pipe (`|`) command to send the output directly to the `rwuniq` command. However, one problem with generating such temporary files is that they slow down the analysis. The `rwfilter` command would have written all the data to disk, and then the subsequent `rwuniq` command would have read the data back from disk. Using UNIX pipes to pass records from one process to another skips the time-consuming steps of writing and reading data, speeding this up considerably. The SiLK tools can operate concurrently, using memory (when possible) to pass data between them.

Additional techniques for improving SiLK performance are described in Chapter 9.

Example 2.6 shows the output from this series of SiLK commands.

```
<1>$ rwfilter flows.rw --bytes=0-300 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >low-byte.txt
<2>$ cat low-byte.txt
              sTime|   type|   Records|
2015/06/17T14:00:00|     in|      1449|
2015/06/17T14:00:00|    out|      1500|
2015/06/17T14:00:00|  inweb|        12|
2015/06/17T14:00:00| outweb|      8339|
<3>$ rwfilter flows.rw --bytes=300-100000 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >medium-byte.txt
<4>$ cat medium-byte.txt
              sTime|   type|   Records|
2015/06/17T14:00:00|     in|        66|
2015/06/17T14:00:00|    out|        96|
2015/06/17T14:00:00|  inweb|       346|
2015/06/17T14:00:00| outweb|     10051|
<5>$ rwfilter flows.rw --bytes=100000- --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >high-byte.txt
<6>$ cat high-byte.txt
              sTime|   type|   Records|
2015/06/17T14:00:00|    out|         3|
2015/06/17T14:00:00| outweb|         2|
```

Example 2.6: Characterizing flow byte counts with `rwuniq`

Help with `rwuniq` Type `rwuniq --help` for a full list of parameters.

Type `man rwuniq` for a detailed description of the `rwuniq` command and its parameters.

Other useful `rwuniq` options

- The `--value` parameter specifies which flow attributes are counted for a time bin. In addition to counting bytes, `rwuniq` can count records, packets, and source and destination IP addresses.
- Flow records need not be sorted before being passed to `rwuniq`. If the records are sorted in the same order as indicated by the `--fields` parameter to `rwuniq`, using the `--presorted-input` parameter may reduce memory requirements for `rwuniq`.

Finding the Most Commonly-Used Protocols With `rwstats`

Another way to characterize network flows is by protocol usage. By looking at the most commonly-used protocols, we can get a sense of what types of traffic the network carries. Normal network traffic is TCP (protocol 6) and UDP (protocol 17), with some ICMP (protocol 1). Networks with more confidential traffic may use IPSEC (with AH [protocol 50] and ESP [protocol 51]) or site-specific encryption (protocol 99).

Use the `rwstats` command to identify the 10 most common protocols associated with traffic into and out of the IP address of interest. `rwstats` groups records into time bins by field (or fields), similar to `rwuniq`. However, `rwstats` can list the top N or bottom N bins and compute summary percentages for each item.

```
rwstats --fields=protocol --count=10 flows.rw
```

- `--fields=protocol` counts the records that carry traffic with each protocol.
- `--count=10` computes statistics for the 10 bins with the most common protocols.
- `flows.rw` is the name of the file containing SiLK network flow records.

Example 2.7 shows the output from this command.

```
<1>$ rwstats --fields=protocol --count=10 flows.rw
INPUT: 21864 Records for 3 Bins and 21864 Total Records
OUTPUT: Top 10 Bins by Records
pro|   Records|  %Records|   cumul_%|
  6|     18854| 86.233077| 86.233077|
 17|      2909| 13.304976| 99.538053|
  1|       101|  0.461947|100.000000|
```

Example 2.7: Finding the top protocols with `rwstats`

Notice that the output lists just three protocols, not ten. This is because only three protocols were used during the time period of interest. `rwstats` also computes the number of records for each protocol and summarizes the percentage of traffic for each protocol.

- Protocol 6 (Transmission Control Protocol, or TCP) makes up approximately 86% of the traffic; it is used by popular applications such as the World Wide Web, email, remote administration, and file transfer programs.
- Protocol 17 (User Datagram Protocol, or UDP) makes up approximately 13% of the traffic; it is used by the Domain Name System (DNS), the Routing Information Protocol (RIP), the Simple Network Management Protocol (SNMP), and the Dynamic Host Configuration Protocol (DHCP). Voice and video is also transmitted using UDP.
- Protocol 1 (Internet Control Message Protocol, or ICMP) makes up less than 1% of the traffic; it is used by network devices (such as routers) to transmit error messages and other information.

Help with `rwstats`. Type `rwstats --help` for a full list of parameters.

Type `man rwstats` for a detailed description of the `rwstats` command and its parameters.

Other useful `rwstats` options. Each call to `rwstats` *must* include exactly one of the following:

- a key containing one or more fields via the `--fields` parameter and an option to determine the number of key values to show via `--count` (shown in Example 2.7), `--percentage`, or `--threshold`
- one of the summary parameters (`--overall-stats` or `--detail-proto-stats`)

The summary parameters for `rwstats` offer a lot of information on network traffic. When invoked with `--overall-stats`, `rwstats` looks at all the traffic together, calculating profiles of bytes per flow, packets per flow, and bytes per packet within each flow. The profiles include numerical values for minimum and maximum values, quartiles, and quartile ranges. Following the numerical values, `rwstats` presents a histogram for each profile, with the data in ten bins to cover the distribution between 0 and the maximum value, with the top bin covering values up to 2^{32}, the maximum possible. Example 2.8 shows the output of `rwstats --overall-stats`.

For a detailed protocol-by-protocol summary, using `--detail-proto-stats` with a list of protocol numbers augments the overall summary with a similar summary for each protocol. The output can be long since `rwstats` generates a set of histograms for each protocol. In Example 2.9, Command 1 shows the start of the overall summary generated by `--detail-proto-stats`, with values identical to those shown in Example 2.8. Command 2 shows the profile generated for just the TCP flow records.

`rwstats` also has a `--values` parameter that allows you to specify a different set of aggregate values to profile when evaluating highest or lowest N bins. These aggregate values include `records` (the default), `bytes`, `packets`, and `distinct:` followed by any flow record field except `icmpTypeCode`. This parameter does not affect the summarizing parameters. The first counting characteristic listed after the `--values` parameter is the one used to calculate the percentages associated with each bin.

Example 2.10 gives two ways to use the `rwstats --values` parameter to expand on the top-N protocols analysis (shown in Example 2.7):

1. Command 1 uses the `--values=bytes` parameter to change the value counted per bin from the default (records) to aggregate bytes in all flows. While this does not change the order of bins in this example, it does show that the volume of transferred data from TCP is a much greater fraction of the total than the count of flows.

```
<1>$ rwstats --overall-stats flows.rw
FLOW STATISTICS--ALL PROTOCOLS:  21864 records
*BYTES min 51; max 210540
  quartiles LQ 126.33764 Med 232.74008 UQ 4906.22492 UQ-LQ 4779.88728
   interval_max|count<=max|%_of_input|   cumul_%|
             40|         0|  0.000000|  0.000000|
             60|        46|  0.210392|  0.210392|
            100|       409|  1.870655|  2.081046|
            150|      9513| 43.509879| 45.590926|
            256|      1235|  5.648555| 51.239480|
           1000|      1409|  6.444383| 57.683864|
          10000|      8723| 39.896634| 97.580498|
         100000|       524|  2.396634| 99.977131|
        1000000|         5|  0.022869|100.000000|
     4294967295|         0|  0.000000|100.000000|
*PACKETS min 1; max 2232
  quartiles LQ 1.51217 Med 3.15094 UQ 13.33673 UQ-LQ 11.82456
   interval_max|count<=max|%_of_input|   cumul_%|
              3|     10844| 49.597512| 49.597512|
              4|       583|  2.666484| 52.263996|
             10|      3662| 16.748994| 69.012989|
             20|      3923| 17.942737| 86.955726|
             50|      1401|  6.407794| 93.363520|
            100|       684|  3.128430| 96.491950|
            500|       714|  3.265642| 99.757592|
           1000|        41|  0.187523| 99.945115|
          10000|        12|  0.054885|100.000000|
     4294967295|         0|  0.000000|100.000000|
*BYTES/PACKET min 40; max 1251
  quartiles LQ 52.54317 Med 65.96176 UQ 119.25022 UQ-LQ 66.70705
   interval_max|count<=max|%_of_input|   cumul_%|
             40|        62|  0.283571|  0.283571|
             44|        17|  0.077753|  0.361325|
             60|     10089| 46.144347| 46.505671|
            100|      5126| 23.444932| 69.950604|
            200|      5735| 26.230333| 96.180937|
            400|       718|  3.283937| 99.464874|
            600|        97|  0.443652| 99.908525|
            800|        13|  0.059458| 99.967984|
           1500|         7|  0.032016|100.000000|
     4294967295|         0|  0.000000|100.000000|
```

Example 2.8: Finding overall traffic profile `rwstats --overall`

```
<1>$ rwstats --detail-proto-stats=6 flows.rw \
| head -3
FLOW STATISTICS--ALL PROTOCOLS:  21864 records
*BYTES min 51; max 210540
  quartiles LQ 126.33764 Med 232.74008 UQ 4906.22492 UQ-LQ 4779.88728
<2>$ rwstats --detail-proto-stats=6 flows.rw \
| tail -n +41

FLOW STATISTICS--PROTOCOL 6:  18854/21864 records
*BYTES min 80; max 210540
  quartiles LQ 129.61289 Med 900.94675 UQ 5678.05941 UQ-LQ 5548.44652
   interval_max|count<=max|%_of_proto|   cumul_%|
             40|         0|  0.000000|  0.000000|
             60|         0|  0.000000|  0.000000|
            100|        62|  0.328843|  0.328843|
            150|      7853| 41.651639| 41.980482|
            256|       340|  1.803331| 43.783812|
           1000|      1352|  7.170892| 50.954705|
          10000|      8719| 46.244829| 97.199533|
         100000|       524|  2.779251| 99.978784|
        1000000|         4|  0.021216|100.000000|
     4294967295|         0|  0.000000|100.000000|
*PACKETS min 2; max 1474
  quartiles LQ 1.78636 Med 5.57930 UQ 15.23493 UQ-LQ 13.44858
   interval_max|count<=max|%_of_proto|   cumul_%|
              3|      7915| 41.980482| 41.980482|
              4|       561|  2.975496| 44.955978|
             10|      3613| 19.163042| 64.119020|
             20|      3916| 20.770128| 84.889148|
             50|      1399|  7.420176| 92.309324|
            100|       684|  3.627877| 95.937202|
            500|       714|  3.786995| 99.724196|
           1000|        41|  0.217460| 99.941657|
          10000|        11|  0.058343|100.000000|
     4294967295|         0|  0.000000|100.000000|
*BYTES/PACKET min 40; max 1251
  quartiles LQ 51.50445 Med 59.13846 UQ 122.73994 UQ-LQ 71.23549
   interval_max|count<=max|%_of_proto|   cumul_%|
             40|        62|  0.328843|  0.328843|
             44|        17|  0.090167|  0.419009|
             60|      9880| 52.402673| 52.821682|
            100|      3038| 16.113292| 68.934974|
            200|      5022| 26.636258| 95.571232|
            400|       718|  3.808210| 99.379442|
            600|        97|  0.514480| 99.893922|
            800|        13|  0.068951| 99.962873|
           1500|         7|  0.037127|100.000000|
     4294967295|         0|  0.000000|100.000000|
```

Example 2.9: Summarizing traffic with one protocol via `rwstats --detail-proto-stats`

2. Command 2 adds a count of unique source IP addresses of flows (`--values=bytes,distinct:sip`) to the byte count in Command 1. Since `bytes` is listed first, bin order and percentages are still computed using the aggregate byte count.

The results show that this traffic comes from only a few sources, which gives insight into the traffic pattern for the host of interest.

```
<1>$ rwstats --fields=protocol --values=bytes --count=10 \
   flows.rw
INPUT: 21864 Records for 3 Bins and 33313611 Total Bytes
OUTPUT: Top 10 Bins by Bytes
pro|               Bytes|    %Bytes|   cumul_%|
  6|            32710148| 98.188539| 98.188539|
 17|              438235|  1.315483| 99.504023|
  1|              165228|  0.495977|100.000000|
<2>$ rwstats --fields=protocol --values=bytes,distinct:sip \
   --count=10 flows.rw
INPUT: 21864 Records for 3 Bins and 33313611 Total Bytes
OUTPUT: Top 10 Bins by Bytes
pro|               Bytes|sIP-Distin|    %Bytes|   cumul_%|
  6|            32710148|         5| 98.188539| 98.188539|
 17|              438235|         2|  1.315483| 99.504023|
  1|              165228|         2|  0.495977|100.000000|
```

Example 2.10: Profiling protocol volumes with `rwstats --values`

Comparing `rwstats` to `rwuniq`

`rwstats` in top or bottom mode and `rwuniq` have much in common, especially since SiLK version 3.0.0. An analyst can perform many tasks with either tool. Some guidelines follow for choosing the tool that best suits a task. Generally speaking, `rwstats` is the workhorse data description tool, but `rwuniq` does have some features that are absent from `rwstats`.

- Like `rwcount`, `rwstats` and `rwuniq` assign flows to bins. For each value of a key, specified by the tool with the `--fields` parameter, a bin summarizes counts of flows, packets, or bytes, or some other measure determined by the analyst with the `--values` parameter. `rwuniq` displays one row of output for every bin except those not achieving optional thresholds specified by the analyst. `rwstats` displays one row of output for each bin in the top N or bottom N, where N is determined directly by the `--count` parameter or indirectly by the `--threshold` or `--percentage` parameters.
- If `rwstats` or `rwuniq` is initiated with multiple counts in the `--values` parameter, the first count is the primary count. `rwstats` can apply a threshold only to the primary count, while `rwuniq` can apply thresholds to any or several counts.
- For display of all bins, `rwuniq` is easiest to use. However, a similar result can be obtained with `rwstats --threshold=1`. `rwstats` will run more slowly than `rwuniq` because it must sort the bins by their summary values.
- `rwstats` always sorts bins by their primary count. `rwuniq` optionally sorts bins by their key.

- `rwstats` normally displays the percentage of all input traffic accounted for in a bin, as well as the cumulative percentage for all the bins displayed so far. This output can be suppressed with `--no-percents` to be more like `rwuniq` or when the primary count is not bytes, packets, or records.
- `rwuniq` has two counts that are not available with `rwstats`: `sTime-Earliest` and `eTime-Latest`.
- Network traffic frequently can be described as exponential, either increasing or decreasing. `rwstats` is good for looking at the main part of the exponential curve, or the tail of the curve, depending on which is more interesting. `rwuniq` provides more control of multi-dimensional data slicing, since its thresholds can specify both a lower bound and an upper bound. `rwuniq` will be better at analyzing multi-modal distributions that are commonly found when the x-axis represents time.

2.2.6 Characterize Traffic by Time Period With `rwcount`

A typical network analysis will examine network traffic by time period to see how it varies throughout the event of interest. Unusual volumes of traffic, changes in byte and packet counts, and other deviations from normal activity can help you to figure out what is causing the event to occur.

The `rwcount` command captures network activity that occurs during the time interval (or bin) that you specify. It counts the number of records, bytes, and packets for flows occurring during a bin's assigned time period. You can then view these counts in a terminal window or graph them in a plotting package such as gnuplot, a spreadsheet package such as Microsoft Excel, or another analysis tool.

Our analysis will examine network traffic to and from the target IP address during the time period of interest. We will use `rwcount` to show this activity in ten-minute time bins.

```
rwcount --bin-size=600 flows.rw
```

- `--bin-size` specifies the time bin in seconds. In this example, the time bin is 600 seconds or ten minutes.
- `flows.rw` contains the network flow records of interest.

Example 2.11 shows the output from this command. It counts the flow volume information gleaned from the `flows.rw` file by ten-minute bins.

```
<1>$ rwcount --bin-size=600 flows.rw
               Date|        Records|                Bytes|          Packets|
2015/06/17T14:00:00|         466.00|             798757.00|          3423.00|
2015/06/17T14:10:00|         394.00|             104668.00|          1622.00|
2015/06/17T14:20:00|         382.43|             104159.18|          1621.86|
2015/06/17T14:30:00|         393.57|             107100.82|          1670.14|
2015/06/17T14:40:00|        9335.01|           15559931.61|        191709.67|
2015/06/17T14:50:00|       10885.11|           16541697.17|        187619.55|
2015/06/17T15:00:00|           7.70|              75830.56|           897.45|
2015/06/17T15:10:00|           0.17|              21466.66|           383.33|
```

Example 2.11: Counting Bytes, Packets and Flows with Respect to Time

By default, `rwcount` produces the table format shown in Example 2.11.

- The first column is the timestamp for the earliest moment in the bin.
- The net three columns show the number of flow records, bytes, and packets counted in the bin.

Examining Bytes, Packets, and Flows

Counting by bytes, packets, and flows can reveal different traffic characteristics. As noted at the beginning of this manual, the majority of traffic crossing wide area networks has very low packet counts. However, this traffic, by virtue of being so small, does not make up a large volume of bytes crossing the enterprise network. Certain activities, such as scanning and worm propagation, are more visible when considering packets, flows, and various filtering criteria for flow records.

The traffic into and out of the IP address of interest (captured in the file `flows.rw`) jumps significantly during the ten-minute time bins `2015/06/17T14:40:00` and `2015/06/17T14:50:00`. Byte, packet, and record counts all rise during this 20-minute time period.

Examining Traffic Over a Period of Time

`rwcount` is used frequently to provide graphs showing activity over long periods of time, giving a visual representation of shifts in network traffic. Count data can be read by most plotting (graphing) applications.

The data from Example 2.11 is plotted using Microsoft Excel in Figure 2.3. The traffic spike that we saw in the tabular data shows up clearly in the plots on the left-hand side of this figure.

For a more detailed look at network activity during this time period, we can change the `--bin-size` from 600 seconds (ten minutes) to 60 seconds (one minute).

```
rwcount --bin-size=60 flows.rw
```

Plots of this data are shown on the right-hand side of Figure 2.3. Looking at the data on a minute-by-minute basis shows the variation in data flows during this event.

Hint 2.8: Data Resolution Versus Bin Size

Whether you use a larger bin size or smaller bin size depends on your data. Smaller bin sizes provide more data points to capture subtleties in traffic. If the bin size is too small, however, it becomes harder to spot trends in the data. Larger bin sizes make it easier to spot regular traffic patterns. If the bin size is too large, however, there will not be enough resolution in the data to see what is happening on your network at a given point in time.

Help with `rwcount`

Type `rwcount --help` for a full list of parameters.

Type `man rwcount` for a detailed description of the `rwcount` command and its parameters.

Other Useful `rwcount` Information

Keep the following in mind when using `rwcount`.

- The default bin size is 30 seconds.
- Bin counts that have zero flows, packets, and bytes can be suppressed by the `--skip-zeroes` option to reduce the length of the listing. However, do not skip rows with zero flows if the output is being passed to a plotting program; if they are, those data points will not be plotted.

2.2.7 Sort Flow Records With `rwsort`

Sorting flow records can help you to organize them according to protocol, IP address, start time, and other attributes. Use the `rwsort` command to sort binary flow records according to the value of the field(s) you select.

`rwsort` is a high-speed sorting tool for SiLK flow records. It reads binary SiLK flow records from a file (or standard input) and outputs the same records in a user-specified order. The output records are also in binary (non-text) format and are not human-readable without interpretation by another SiLK tool such as `rwcut`. `rwsort` is faster than the standard UNIX `sort` command, handles flow record fields directly with understanding of the fields' types, and is capable of handling very large numbers of SiLK flow records provided sufficient memory and storage are available.

The following example sorts the network flow records in `flows.rw` by byte count, destination IP, and protocol from the highest to the lowest value in each field, then displays the first ten records.

```
rwsort flows.rw --fields=dip,protocol,bytes --reverse
| rwcut --fields=dip,protocol,bytes,stime --num-recs=10
```

- `--fields` specifies the sort order. It identifies the fields that are used as sort keys and specifies their precedence. In this example, `rwsort` first sorts the records by destination IP address (`dip`), then protocol (`protocol`), then byte count (`bytes`).
- By default, `rwsort` sorts from the lowest to the highest values of each sort key. `--reverse` sorts the records from the highest to the lowest values.
- The file `flows.rw` contains the SiLK record files to be sorted.
- The Unix pipe command (`|`) sends the output of the `rwsort` command to the `rwcut` command. The `rwcut` command and its parameters are described in Section 2.2.3.

Example 2.12 shows the results of this command. The records are first sorted from the highest destination IP address to the lowest. They are then sorted according to their protocols, then their sizes in bytes. This gives you an idea of the volume and types of traffic associated with the destination IPs.

```
<1>$ rwsort flows.rw --fields=dip,protocol,bytes --reverse \
| rwcut --fields=dip,protocol,bytes,stime --num-recs=10
            dIP|pro|     bytes|                  sTime|
  216.207.68.32|  6|       960|2015/06/17T14:53:15.707|
  216.207.68.32|  6|       960|2015/06/17T14:54:30.604|
  216.207.68.32|  6|       120|2015/06/17T14:54:14.405|
  216.207.68.32|  6|       120|2015/06/17T14:55:29.333|
  209.66.102.50|  6|       960|2015/06/17T14:55:26.586|
  209.66.102.50|  6|       960|2015/06/17T14:56:42.465|
  209.66.102.50|  6|       120|2015/06/17T14:57:41.186|
  209.66.102.50|  6|       120|2015/06/17T14:56:25.264|
  208.206.41.61|  6|       960|2015/06/17T14:58:20.666|
  208.206.41.61|  6|       960|2015/06/17T14:46:17.427|
```

Example 2.12: Sorting by Destination IP Address, Protocol, and Byte Count

Behavioral Analysis with `rwsort`, `rwcut`, and `rwfilter`

A behavioral analysis of protocol activity relies heavily on basic `rwcut` and `rwfilter` parameters. The analysis requires the analyst to have a thorough understanding of how protocols are meant to function. Some concept of baseline activity for a protocol on the network is needed for comparison.

To monitor the behavior of protocols, first take a sample of a particular protocol. Use `rwsort --fields=sTime`, and convert the results to ASCII text with `rwcut`. To produce byte and packet fields only, try `rwcut` with `--fields=bytes` and `--fields=packets`. Then, perform the UNIX commands `sort` and `uniq -c`.

Cutting in this manner (sorting by field or displaying select fields) can answer a number of questions:

1. Is there a standard bytes-per-packet ratio?
2. Do any bytes-per-packet ratios fall outside the baseline?
3. Do any sessions' byte counts, packet counts, or other fields fall outside the norm?

There are many such questions to ask, but keep the focus of exploration on the behavior being examined. Chasing down weird cases is tempting but can add little to your understanding of general network behavior.

Help with `rwsort`

Type `rwsort --help` for a full list of parameters.

Type `man rwsort` for a detailed description of the `rwsort` command and its parameters.

Other Useful `rwsort` Information

Keep the following in mind when using `rwsort`.

- Sort keys can be specified by field numbers as well as field names; see Table 1.1 for a complete list.

- Sort keys can be specified in any order. For example, `--fields=1,3` results in flow records being sorted by source IP address (1) and by source port (3) for each source IP address. `--fields=3,1` results in flow records being sorted by source port and by source IP address for each source port. (Since flow records are not always entered into the repository in the order in which they were initiated, analyses often involve sorting by start time at some point.)
- `rwsort` can also be used to sort multiple SiLK record files. If the flow records in the input files are already ordered in each file, using the `--presorted-input` parameter can improve efficiency significantly by just merging the files.
- If `rwsort` is processing large input files, disk space in the default temporary system space may be insufficient or not located on the fastest storage available. To use an alternate space, specify the `--temp-directory` parameter with an argument specifying the alternate space. This may also improve data privacy by specifying local, private storage instead of shared storage. See Chapter 9 for more information on optimizing SiLK analytics.

2.2.8 Use IPsets to Gather IP Addresses

Up to this point, our single-path analysis has focused on selecting, storing, and examining flow records. However, another common goal of single-path analysis is to compile lists of IP addresses that exhibit criteria of interest to the analyst. This section will continue our analysis by gathering and summarizing single-path criteria using named sets of IP addresses, or *IPsets*.

Create IPsets With `rwset` and `rwsetbuild`

`rwset` and `rwsetbuild` are two SiLK tools for creating sets of IP addresses (IPsets). `rwset` creates sets from flow records. `rwsetbuild` creates them from lists of IP addresses in text files. Expanding on the profiling in Section 2.2.5, `rwfilter` can be used to profile network flows by bytes. When combined, `rwset` and `rwfilter` summarize the IP addresses that exhibit byte-threshold profiles to files with descriptive names.

```
rwfilter flows.rw --bytes=0-300 --pass=stdout \
| rwset --any-file=low-byte.set
```

Parameters for the `rwfilter` command include the following:

- `flows.rw` contains the network flow records of interest.
- `--bytes=0-300` specifies the range of byte counts for selecting records (0-300 for this example).
- `--pass=stdout` sends all records that pass the filter to standard output.

Parameters for the `rwset` command include the following:

- `--any-file=low-bytes.set` specifies source and destination IP addresses from flow records with a range of 0-300 bytes to the IPset file `low-bytes.set`. Because `--any-file` was used above, the IPset file will include the IP address itself as well as any IP addresses that communicated with it.

```
<1>$ rwfilter flows.rw --bytes=0-300 --pass=stdout \
| rwset --any-file=low-byte.set
<2>$ file low-byte.set
low-byte.set: SiLK, IPSET v2, Little Endian, LZO compression
```

Example 2.13: Using `rwset` to Gather IP Addresses

Example 2.13 shows the output from this series of `rwfilter` and `rwset` commands.

Analysis requiring defined IP addresses should use the `rwsetbuild` tool. `rwsetbuild` creates SiLK IPsets from textual input, including canonical IP addresses, CIDR notation, and IP ranges. This approach is useful for creating whitelists and blacklists of IP addresses that may reside in network flow records presently or in the future.

```
rwsetbuild --ip-ranges servers.txtservers.set
```

Parameters for the `rwsetbuild` command include the following:

- `--ip-ranges` specifies allowing the textual input file to contain IP ranges.
- `servers.txt` specifies the textual input file name.
- `servers.set` specifies the binary IPset output file name.

Example 2.14 shows the output from the `rwsetbuild` command.

```
<1>$ cat servers.txt
# Text file of servers
192.168.2.1 # Single
192.168.3.0/24 # CIDR
192.168.4.1-192.168.4.128 # IP range
<2>$ rwsetbuild --ip-ranges servers.txt servers.set
<3>$ file servers.set
servers.set: SiLK, IPSET v2, Little Endian, LZO compression
```

Example 2.14: Using `rwsetbuild` to Gather IP Addresses

Help with `rwset` and `rwsetbuild`

Type `rwset --help` or `rwsetbuild --help` for a full list of parameters.

Type `man rwset` or `man rwsetbuild` for a detailed description of these commands and their parameters.

Other Useful `rwset` and `rwsetbuild` Options

Keep the following in mind while using the `rwset` command:

- `rwset` can assign IP addresses to IPsets by source, destination, and both source and destination simultaneously.
- `rwset` and `rwsetbuild` can read input from files on disk or standard input (`stdin`).
- `rwsetbuild` supports SiLK IP address wildcard notation (10.x.1-2.4,5). This notation is not supported when the `--ip-ranges` switch is specified.

Display IP Addresses, Counts, and Network Information With `rwsetcat`

Single-path analysis often requires the IP addresses in an IPset to be counted and displayed. This gives you an opportunity to inspect the IP addresses that met specified analytic criteria, such as behavior and network topology. `rwsetcat` can display the IP addresses in an IPset, count the number of IP addresses, display information about the network, and show minimum and maximum IP addresses as well as other summary data for the IPset.

We will continue our analysis of the low byte IP addresses from Section 2.2.8 by counting, listing, and computing summary statistics for the IP addresses in the `low-bytes.set` IPset file.

To count the number of IP addresses that exhibited 0-300 byte flow records with the IP address `192.168.70.10`, enter the following command:

```
rwsetcat --count-ips low-byte.set
```

Parameters for the `rwsetcat` command include the following:

- `--count-ips` specifies counting the number of IP addresses.
- `low-byte.set` specifies the binary IPset file for counting (containing IP addresses that exhibited 0-300 byte flow records with `192.168.70.10`.)

Example 2.15 shows the count of IP addresses contained in `low-byte.set`.

```
<1>$ rwsetcat --count-ips low-byte.set
574
```

Example 2.15: Using `rwsetcat` to Count Gathered IP Addresses

Although general counting is helpful, an analysis commonly requires additional context regarding the networks and hosts contained in the IPset. `rwsetcat` prints analyst-specified subnet ranges and the number of hosts in each subnet.

To summarize the /24 networks contained in `low-byte.set`:

```
rwsetcat --network-structure=24 low-byte.set
```

Parameters for the `rwsetcat` command include the following:

- `--network-structure` groups IP addresses by specified structure and prints the number of hosts

- `low-byte.set` specifies the binary IPset file for counting (containing IP addresses that exhibited 0-300 byte flow records with `192.168.70.10`.)

Example 2.16 shows the first four /24 networks contained in `low-byte.set` and their respective host counts.

```
<1>$ rwsetcat --network-structure=24 low-byte.set | head -n 4
        4.2.0.0/24| 1
        6.7.1.0/24| 1
        8.1.7.0/24| 1
       10.0.20.0/24| 1
```

Example 2.16: Using `rwsetcat` to Print Networks and Host Counts

Complete statistical summaries are also common during an analysis and can be printed with `rwsetcat`. Our previous /24 summary only prints the specified CIDR range and would require iterative commands to determine multiple CIDR network ranges that may be contained in an IPset. Therefore, `rwsetcat` provides the `--print-statistics` switch for full statistical summaries of an IPset.

```
rwsetcat --print-statistics low-byte.set
```

Parameters for the `rwsetcat` command include the following:

- `--print-statistics` specifies printing a statistical summary of IP addresses contained in an IPset.
- `low-byte.set` specifies the binary IPset file for counting (containing IP addresses that exhibited 0-300 byte flow records with `192.168.70.10`.)

Example 2.17 shows the statistical summary of IP addresses in the `low-byte.set` file.

```
<1>$ rwsetcat --print-statistics low-byte.set
Network Summary
        minimumIP =         4.2.0.58
        maximumIP =    216.207.68.32
               574 hosts (/32s),    0.000013% of 2^32
                87 occupied /8s,   33.984375% of 2^8
               381 occupied /16s,   0.581360% of 2^16
               521 occupied /24s,   0.003105% of 2^24
               551 occupied /27s,   0.000411% of 2^27
```

Example 2.17: Using `rwsetcat` to Print IP Address Statistical Summaries

Help with `rwsetcat`

Type `rwsetcat --help` for a full list of parameters.

Type `man rwsetcat` for a detailed description of this command and its parameters.

Other Useful `rwsetcat` Options

Keep the following in mind while using the `rwsetcat` command:

- `rwsetcat` can print CIDR blocks without specifying a desired network mask. `rwsetcat` will group sequential IPs into the largest possible CIDR block and prints individual IP addresses. This switch cannot be combined with the `--network-structure` switch.
- The `--network-structure` switch supports multiple CIDR masks for a single command execution.

2.2.9 Resolve IP Addresses to Domain Names With `rwresolve`

Use the `rwresolve` command to display the hostnames associated with the IP addresses of interest to our network analysis. This command performs a reverse Domain Name Service (DNS) lookup on a list of IP addresses to retrieve their host names. If the lookup is successful, it prints the name of the host; if not, it prints the IP address. If an IP address resolves to multiple host names, it prints the first one found. The result is a human-readable list of hostnames that is useful for further investigation and analysis.

`rwresolve` takes delimited text as input, not binary flow records. It is designed for use with the `rwcut` command, although it can be used with any SiLK tool that produces delimited text.

Hint 2.9: Use `rwresolve` with Small Datasets

Since performing reverse DNS lookups is a time-consuming process, we strongly recommend that you use `rwresolve` only on small datasets.

```
rwcut --fields=1,1 flows.rw | rwresolve --ip-field=2
```

This command first uses the `rwcut` command to generate a list of source and destination IP addresses (`--fields=1,1`). It redirects the resulting output to the `rwresolve` command, which looks up the host names associated with the destination IP addresses.

Example 2.18 shows the default behavior of `rwresolve`. The output of the `rwcut` command is passed as input to the `rwresolve` command. By default, `rwresolve` will attempt to resolve source and destination IP addresses without the `--ip-field` option. However, this example shows that DNS was not able to resolve all source IP addresses.

`rwresolve` supports the *c-ares* and *adns* asynchronous DNS libraries and will automatically select what is available when the SiLK tool suite is installed. The *getnameinfo* and *gethostbyaddr* C libraries are also supported, however, these may impact DNS resolution speed. Analysts can select the desired resolver using the `--resolver` switch.

Example 2.19 shows how to display the destination IP address field with `rwresolve`. It is important to note that field 2 is the default position for destination IP addresses in SiLK network flow records. Therefore, if analysts decide to append a source IP address to `rwcut` output as a method for displaying the IP and hostname (such as `rwcut --fields=1-12,1`), the `--ip-fields=13` would be a required option for `rwresolve` to determine that the 13th flow record field should be resolved.

```
<1>$ rwcut --fields=sip,dip,sport,dport,protocol --num-recs=10 \
   --ipv6-policy=ignore flows.rw \
| rwresolve
                 sIP|                       dIP|sPort|dPort|pro|
          10.0.40.83|  divpx02un00001.div1.net|53981| 8082|  6|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|58887| 17|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|55004| 17|
          10.0.40.83|  divpx02un00001.div1.net|53982| 8082|  6|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|64408| 17|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|57734| 17|
          10.0.40.83|  divpx02un00001.div1.net|53983| 8082|  6|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|63770| 17|
soca1202un1.div0.net|  divpx02un00001.div1.net|   53|53374| 17|
          10.0.40.83|  divpx02un00001.div1.net|53984| 8082|  6|
```

Example 2.18: Looking Up Source and Destination Hostnames with `rwresolve`

```
<1>$ rwcut --fields=sip,dip,sport,dport,protocol --num-recs=10 \
   --ipv6-policy=ignore flows.rw \
| rwresolve --ip-fields=2
       sIP|                       dIP|sPort|dPort|pro|
10.0.40.83|  divpx02un00001.div1.net|53981| 8082|  6|
10.0.40.20|  divpx02un00001.div1.net|   53|58887| 17|
10.0.40.20|  divpx02un00001.div1.net|   53|55004| 17|
10.0.40.83|  divpx02un00001.div1.net|53982| 8082|  6|
10.0.40.20|  divpx02un00001.div1.net|   53|64408| 17|
10.0.40.20|  divpx02un00001.div1.net|   53|57734| 17|
10.0.40.83|  divpx02un00001.div1.net|53983| 8082|  6|
10.0.40.20|  divpx02un00001.div1.net|   53|63770| 17|
10.0.40.20|  divpx02un00001.div1.net|   53|53374| 17|
10.0.40.83|  divpx02un00001.div1.net|53984| 8082|  6|
```

Example 2.19: Looking Up Destination Hostnames with `rwresolve`

Help with `rwresolve`

Type `rwresolve --help` for a full list of parameters.

Type `man rwresolve` for a detailed description of this command and its parameters.

Figure 2.3: Displaying `rwcount` Output Using 10-Minute and 1-Minute Bins

2.3 Situational Awareness and Single-Path Analysis

Situational awareness is the perception of environmental elements and events with respect to time or space, the comprehension of their meaning, and the projection of their future status.[8] What does this mean for cybersecurity? Analysts generally need to find answers to the following questions about their networks:

- What is currently present?
- What is currently threatening?
- What potential developments (either aggressive or defensive) might be expected?

This information can then be used as a basis for making decisions about the operation and defense of your cyber infrastructure.

We'll examine how to apply SiLK's single-path analysis workflow to gain better situational awareness by looking at examples. Our example cybersecurity analyst named Arthur is asked to examine the web traffic on his organization's network. Arthur's supervisor is concerned about abnormal traffic and is worried that the organization's web site might be maliciously redirected. Two possible ways of doing this redirection are direct attacks on the web server and manipulation of the DNS cache to redirect otherwise-benign content references.

Arthur can take several steps to gain awareness of the organization's web service and the threats to that web service. We'll discuss them later in this section.

2.3.1 Components of Situational Awareness

Situational awareness for cybersecurity has four main components.

- Know what should be (referred to as *Desired awareness*)
- Track what is—i.e., what is actually in use, not just installed (referred to as *Actual awareness*)
- Infer when "what should be" does not match "what is" (referred to as *Differential awareness*)
- Do something about the differences (referred to as *Actionable awareness*)

Desired awareness implies understanding the enterprise's intent for required activities, whether from an infrastructure or a mission-assurance point of view, including

- legitimate users of internal and public-facing systems and devices
- approved software and applications to be installed on network hosts

[8] Endsley, M. R. (1995). Toward a theory of situation awareness in dynamic systems. Human Factors, 37(1), 32-64.

- expected communication patterns from internal systems to external systems and to other internal systems
- specific anticipated changes that are currently happening

Actual awareness implies measured observation of the enterprise network and its traffic to determine the current status, including

- observed devices, software or applications, and users
- which known vulnerabilities exist for the observed devices, software and applications
- how usage of the devices is changing
- usage patterns and cycles associated with systems, devices and users

Differential and actionable awareness most often involve multi-path or exploratory analyses and will be discussed further in Section 4.3 and Section 6.3.

2.3.2 Single-Path Analysis for Desired Awareness: Validate Web and DNS Servers

An analyst can do a lot with single-path analyses to gain desired awareness. Many network functions are identified by specific applications, protocols, or IP addresses. Each of these functions can be addressed via a specific analysis.

In Arthur's case, he knows that the IP address of the primary web server is `10.0.40.21`. He also knows that the organization includes multiple branches, with DNS authoritative and recursive resolvers at IP addresses `192.168.20.58`, `10.0.40.20`, `10.0.20.58`, and `192.168.40.20`. Arthur needs to find out whether these servers are operating as expected. Example 2.20 shows Arthur's analysis to validate the expected servers.

Validate Primary Web Server

Command 1 in Example 2.20 uses a single-path combination of `rwfilter` and `rwstats`. The `rwfilter` call pulls more than three weeks of traffic, but most of that traffic is empty in the sample data. From these records, the `rwfilter` call filters outbound web flows that came from port `443`, the reserved port for https service.

The results from Command 1 list the top 5 web servers that are active on the network. They validate that the organization's main web server, `10.0.40.21`, is the most common web server.

Find Recursive and Authoritative DNS Resolvers

Profiling DNS service is slightly trickier. Arthur now needs to look at inbound and outbound DNS traffic (UDP on port 53) to find recursive and authoritative resolvers. A recursive resolver is a server that generates outbound DNS queries. An authoritative resolver is a server that responds to inbound DNS queries. The four DNS servers should be acting as both recursive and authoritative resolvers if they are operating properly.

Command 2 in Example 2.20 uses `rwfilter` and `rwstats` to look for recursive resolvers (source addresses for flows found by `--type=out`, `--proto=17` (UDP), and `--dport=53`). The call identifies five servers: the four expected, plus a relatively-small set of queries from the host `192.168.40.25`.

Command 3 uses `rwfilter` and `rwstats` to look for authoritative resolvers (destination addresses for flows found by `--type=in`, `--proto=17` (UDP), and `--dport=53`). Only the four expected servers are listed in the results. Since these four appear on both lists, they are acting as both authoritative and recursive resolvers, just as expected. However, Arthur decides that host `192.168.40.25` needs a closer look, since it appears on one list but not the other.

```
<1>$ rwfilter --start=2015/06/01 --end=2015/06/25 --type=outweb \
   --sport=443 --pass=stdout \
| rwstats --fields=sip --values=flows --count=5
INPUT: 728080 Records for 5 Bins and 728080 Total Records
OUTPUT: Top 5 Bins by Records
            sIP|   Records|  %Records|   cumul_%|
     10.0.40.21|    727902| 99.975552| 99.975552|
  192.168.40.24|        97|  0.013323| 99.988875|
  192.168.40.91|        38|  0.005219| 99.994094|
  192.168.40.27|        27|  0.003708| 99.997802|
  192.168.40.92|        16|  0.002198|100.000000|
<2>$ rwfilter --start=2015/06/01 --end=2015/06/25 --type=out \
   --proto=17 --dport=53 --pass=stdout \
| rwstats --fields=sip --values=flows --count=5
INPUT: 10741620 Records for 91 Bins and 10741620 Total Records
OUTPUT: Top 5 Bins by Records
            sIP|   Records|  %Records|   cumul_%|
  192.168.20.58|   4838813| 45.047330| 45.047330|
     10.0.20.58|   2071459| 19.284419| 64.331749|
     10.0.40.20|   1697322| 15.801360| 80.133108|
  192.168.40.20|   1135206| 10.568294| 90.701403|
  192.168.40.25|    247595|  2.305006| 93.006409|
<3>$ rwfilter --start=2015/06/01 --end=2015/06/25 --type=in \
   --proto=17 --dport=53 --pass=stdout \
| rwstats --fields=dip --values=flows --count=10
INPUT: 2832800 Records for 4 Bins and 2832800 Total Records
OUTPUT: Top 10 Bins by Records
            dIP|   Records|  %Records|   cumul_%|
  192.168.40.20|   1558373| 55.011755| 55.011755|
     10.0.40.20|    984275| 34.745658| 89.757413|
  192.168.20.58|    282775|  9.982173| 99.739586|
     10.0.20.58|      7377|  0.260414|100.000000|
```

Example 2.20: Using `rwfilter` and `rwstats` to Profile Web and DNS Services

Profile the Anomalous DNS Server

Example 2.21 shows Arthur's analysis of the anomalous host `192.168.40.25`. To do a quick look at that address's activity, Arthur first has to find out which sensors this address lay behind, to minimize the amount of duplication in later results.

Command 1 calls `rwfilter` to collect outgoing traffic (types `out` and `outweb`) with the source address of this host (`192.168.40.25`). It found three sensors that detected this outgoing traffic: `S3`. `S12`. and `S1`.

Arthur can now put this information to use when he looks at activity associated with `192.168.40.25`. Command 2 calls `rwfilter` to pull all records involving this address from these sensors and then feeds the records to `rwuniq`. The `rwuniq` command breaks out activity by type of flow and by the transport protocol in use. The profile identifies the number of flows and the breadth of activity by counting both the source and destination addresses and ports.

The results of Command 2 show that this host is performing quite a bit of activity besides the DNS queries. It is using a lot of TCP and some ICMP traffic. Its profile is similar to one that is normally expected of a workstation, but the number of hosts and ports it communicates with are both larger than the typical workstation. However, this host does not seem to be further relevant to activity against the web server.

```
<1>$ rwfilter --start=2015/06/01 --end=2015/06/25 \
   --type=out,outweb --saddr=192.168.40.25 --pass=stdout \
| rwuniq --fields=sensor --values=flows
sen|   Records|
 S3|    224254|
S12|     18959|
 S1|    109067|
<2>$ rwfilter --start=2015/06/01 --end=2015/06/25 \
   --sensor=S1,S3,S12 --type=all --any-addr=192.168.40.25 \
   --pass=stdout \
| rwuniq --fields=type,protocol \
   --values=flows,distinct:sip,distinct:dip,distinct:sport,distinct:dport \
   --sort
   type|pro|   Records|sIP-Distin|dIP-Distin|sPort|dPort|
     in|  1|        70|         7|         1|    1|    1|
     in|  6|     76656|         8|         1| 1201|10178|
     in| 17|    248417|        69|         1|    4|16240|
    out|  6|     75544|         1|         8|10179| 1201|
    out| 17|    259080|         1|       142|16286|    5|
  inweb|  6|     18459|         1|         1|    2| 5051|
 outweb|  6|     17656|         1|         1| 5052|    2|
int2int|  1|     23549|        11|         5|    1|   10|
int2int|  6|     28992|         4|         5| 9484| 8793|
int2int| 17|    159099|        68|       142|  940|   25|
```

Example 2.21: Using `rwuniq` to Profile an Address

2.3.3 Single-Path Analysis for Actual Awareness: Examine Network Traffic

Arthur needs to do additional work to more fully validate the web and DNS behavior. This moves from analyzing the network for desired awareness to analyzing it for actual awareness. It also sets up later analyses for differential and actionable awareness, which involve more detailed examinations of network traffic to gain a better understanding of the network's current behavior.

Profile Outbound Web Traffic

Arthur decides to profile outbound web traffic from the organization's main web server (`10.0.40.21`). This will show whether the server is carrying normal web traffic or is behaving in a way that might indicate suspicious activity (e.g., signaling or probing attempts).

In Example 2.22, Command 1 uses `rwfilter` to retrieve outbound web flows (`--type=outweb` and `--sport=443`) for a single day. The query is limited to flows with an average of 60 or more bytes per packet (`--bytes-per=60-`) — that is, flows with substantive data content. The `rwfilter` command displays the number of flows and then outputs those flows to the `rwstats` command.

To compute a statistical summary of the flows, Command 1 calls `rwstats` with the `--detail` parameter, which requires a list of protocols as an argument. Although the results of the `rwstats` call include both histograms and summary statistics, Command 1 calls the UNIX `grep` command to display only the statistics. The results profile the total number of bytes per flow, the number of packets per flow, and the bytes per packet average per flow.

Command 2 is similar to Command 1. It uses `rwfilter` and `rwstats` to profile outbound web flows with less than 60 bytes per packet (`--bytes-per=1-59`) — that is, flows that hold little or no data content.

The results of Commands 1 and 2 show that the two groups of flows are very different. The flows with data involve both more packets and much larger bytes per packet averages. The flows with little to no data are short (1-9 packets) and have small total bytes per flow (40-372 bytes).

These results support Arthur's suspicion that the data-bearing flows are likely normal web traffic and the non-data-bearing flows are likely either signalling or probing. The relatively large number of non-data-bearing flows (more than twice the number of data-bearing flows) further supports his interpretation of signalling or probing.

Whether the behavior of non-data-bearing flows becomes part of differential awareness requires exploratory analysis. However, Arthur knows that it is a behavior that might not be expected by the enterprise.

```
<1>$ rwfilter --start=2015/06/17 --type=outweb --sport=443 \
   --saddress=10.0.40.21 --print-stat --bytes-per=60- \
   --pass=stdout \
| rwstats --detail=6 \
| grep "min"
Files   225.  Read    3164253.  Pass     128452. Fail     3035801.
*BYTES min 391; max 154201
*PACKETS min 3; max 118
*BYTES/PACKET min 82; max 1463

<2>$ rwfilter --start=2015/06/17 --type=outweb --sport=443 \
   --saddress=10.0.40.21 --print-stat --bytes-per=1-59 \
   --pass=stdout \
| rwstats --detail=6 \
| grep "min"
Files   225.  Read    3164253.  Pass     323660. Fail     2840593.
*BYTES min 40; max 372
*PACKETS min 1; max 9
*BYTES/PACKET min 40; max 58
```

Example 2.22: Using `rwfilter` and `rwstats` for Web Actual Awareness

Characterize Traffic Between DNS Resolvers

The next step in Arthur's analysis is to examine traffic between recursive and authoritative DNS resolvers. In Example 2.23, the key set of statistics is the bytes per packet. Command 1 shows how to compute bytes per packet for recursive resolvers. Command 2 shows how to compute bytes per packet for authoritative resolvers.

These values are similar for both the recursive and authoritative resolvers. The results are very consistent with the byte size of normal DNS requests and responses. In several cases, multiple requests (up to 216) went to the authoritative resolver from the same recursive resolver within the active timeout window for flow generation. This multiplicity led to the total byte and packet count statistics showing a wide range. While this is somewhat unusual for an operational network, this data is from an exercise. The tight timeframe of the exercise likely led to the multiplicity of queries. The sources and destinations of the traffic are as expected.

```
<1>$ rwfilter --start=2015/06/17 --type=out --protocol=17 \
   --dport=53 --sipset=aware_dns.set --pass=stdout \
| rwstats --detail=17 \
| grep "min"
*BYTES min 45; max 5370
*PACKETS min 1; max 73
*BYTES/PACKET min 45; max 261

<2>$ rwfilter --start=2015/06/17 --type=in --protocol=17 \
   --dport=53 --dipset=aware_dns.set --pass=stdout \
| rwstats --detail=17 \
| grep "min"
*BYTES min 45; max 15889
*PACKETS min 1; max 216
*BYTES/PACKET min 45; max 261
```

Example 2.23: Using `rwfilter` and `rwstats` for DNS Actual Awareness

Conclusions from the Single-path Analysis for Situational Awareness

The web and DNS examples illustrate the use of SiLK for desired and actual situational awareness through single-path analysis. In both cases, the analysis broke the behavior into cases that could be examined separately to support more detailed analysis.

From the results of this analysis, Arthur concludes that the main web server might be exhibiting behavior that indicates signalling or probing in addition to its normal activities. He's also found a server that is engaging in an anomalous pattern of activity. While these results show some unusual behavior, Arthur needs to perform further analysis to find out whether they represent a threat to the security of the enterprise.

Another approach is to work from behavior that is known to be threatening, by focusing on IDS rules. The next section discusses a specialized tool that supports using IDS rules as a basis for retrieving flow records.

2.3.4 Translate IDS Signatures into `rwfilter` Calls with `rwidsquery`

Traditional intrusion detection depends heavily on the presence of payloads and *signatures*: distinctive packet data that can be used to identify a particular intrusion tool. In general, the SiLK tool suite is intended for examining trends. However, it it can also be used to identify specific intrusion tools. While targeted intrusions are still a threat, tool-based, broad-scale intrusions are more common. Sometimes it is necessary to translate an intrusion signature into SiLK filtering rules; this section describes some standard guidelines to accomplish this task.

To convert signatures, consider the intrusion tool behavior as captured in a signature:

- What service is it targeting? This can be converted to a port number.
- What protocol does it use? This can be converted to a protocol number.
- Does it involve several protocols? Some tools, malicious and benign, will use multiple protocols, such as TCP and ICMP.
- What about packets? Buffer overflows are a depressingly common form of attack and are a function of the packet's size as well as its contents. Identifying a specific packet size can help you to figure out which intrusion tool (or tools) may have been employed for the attack.

Hint 2.10: SiLK Byte Counts Include Packet Headers

When working with packet sizes, remember that the SiLK suite includes packet headers. For example, a 376-byte UDP payload will be 404 bytes long after adding 20 bytes for the IP header and eight bytes for the UDP header.

- How large are sessions? An attack tool may use a distinctive session each time (for example, a session of 14 packets with a total size of 2,080 bytes).

The `rwidsquery` command supports direct translation of rules and alert logs from the SNORT® intrusion detection system (IDS) into `rwfilter` queries. This helps an analyst examine network behavior shortly before, during, and after an alert is generated. Look for possible event triggers as well as behaviors that show the alert to be a false positive.

In Example 2.24, a sample Snort rule from Emerging Threats is used as a basis for querying network flow records. Command 1 shows the Snort rule (which looks for a VOIP attack). Command 2 invokes the `rwidsquery` to translate this rule into a call to `rwfilter`, storing the identified flow records in `out.rw`, while reporting how many records were found. Command 3 uses `rwfileinfo` to show the call to `rwfilter` produced by `rwidsquery`.

While the `rwfilter` calls produced by `rwidsquery` are somewhat redundant, this tool allows for a useful starting point for moving from Snort rules and alerts to network flow records. Initially, the results are likely to contain more than the records directly related to the suspicious activity. However, an analyst can add parameters to the `rwfilter` call generated by `rwidsquery`, by using an empty parameter (`--`) followed by the additional `rwfilter` parameters (shown in Command 2 for adding the `--type` parameter). In this way, flow sizes (bytes or packets), addresses (source or destination), and TCP flag combinations can be added to reduce the extra records retrieved through `rwidsquery`. An analyst can also use additional SiLK commands to profile and eliminate records.

```
<1>$ fmt tmp-rule.txt
alert tcp any any -> any 5060 (msg:"ET VOIP INVITE Message Flood
TCP"; flow:established,to_server; content:"INVITE"; depth:6;
threshold: type both , track by_src, count 100, seconds 60;
reference:url,doc.emergingthreats.net/2003192; classtype:attempted-dos;
sid:2003192; rev:4; metadata:created_at 2010_07_30, updated_at
2010_07_30;)
<2>$ rwidsquery --intype=rule \
   --start=2015/06/17T0 --end=2015/06/17T23 \
   --config-file=./snort.2091101.conf.txt tmp-rule.txt \
   --output-file=out.rw -- --type=in,out --print-stat
Files   460.  Read   16661836.  Pass        514. Fail    16661322.
<3>$ rwfileinfo --fields=command-lines out.rw \
| fmt
out.rw:
  command-lines
                  1  rwfilter --start-date=2015/06/17T0
                  --end-date=2015/06/17T23
                  --stime=2015/06/17T0-2015/06/17T23 --dport=5060
                  --type=in,out --print-stat --pass=out.rw
```

Example 2.24: Using `rwidsquery` for Snort Rule Translation

Getting Help with `rwidsquery`

For a list of `rwidsquery` options, type `rwidsquery --help`.

For a complete discussion of this command, type `man rwidsquery`.

2.4 Summary of SiLK Commands in Chapter 2

Command	Section Name	Page
`rwsiteinfo`	Get a List of Sensors With `rwsiteinfo`	19
`rwfilter`	Choose Flow Records With `rwfilter`	23
`rwcut`	View Flow Records With `rwcut`	27
`rwfileinfo`	Viewing File Information with `rwfileinfo`	29
`rwuniq` and `rwstats`	Profile Flows With `rwuniq` and `rwstats`	31
`rwcount`	Characterize Traffic by Time Period With `rwcount`	39
`rwsort`	Sort Flow Records With `rwsort`	41
`rwset` and `rwsetbuild`	Create IPsets With `rwset` and `rwsetbuild`	43
`rwsetcat`	Display IP Addresses, Counts, and Network Information With `rwsetcat`	45
`rwresolve`	Resolve IP Addresses to Domain Names With `rwresolve`	47
`rwidsquery`	Translate IDS Signatures into `rwfilter` Calls with `rwidsquery`	57

Chapter 3

Case Studies: Basic Single-path Analysis

The previous chapter introduced the process of single-path analysis and covered some of the commands that are used in such analyses. This chapter walks through several detailed cases that serve as examples of single-path analyses.

Upon completion of this chapter you will be able to

- describe a sequence of steps that analysts may use in approaching a task
- apply those steps to several tasks relevant to network traffic
- use SiLK tools to automate the analysis

The case studies in this chapter use the FCCX dataset described in Section 1.8.

3.1 Profile Traffic Around an Event

One view of a network security event is that some specific activity occurs on a particular host at an identified time. In terms of the analysis in this handbook, a host is indicated by its IP address, and time is, at first consideration, associated with a given hour. With this as a starting point, the analyst needs to develop a high-level assessment of possible changes in behavior which then provides a guide to more detailed follow-on assessments. The end goal is to answer some basic questions about the event:

- Did the event impact network performance or services?
- Was the impact sufficient to warrant dedicating resources to respond?

- Was the event malicious?
- Did it demonstrate weaknesses that could enable malicious activity?
- Which entities were involved (both internal and external)?

Frequently, trying to answer such questions in detail involves too much effort. The alternative is to proceed in a staged manner, and at each stage determine if analysis should proceed further. This section describes an initial high-level analysis that can be done rapidly. It helps you to determine whether there could have been some impact from the event, along with a rough feel as to the magnitude of that impact.

Working from the target IP address and the time frame as a start, there are several possible approaches to gaining a high level indication of impact from the event:

- **traffic**—look at traffic on the targeted network and search for shifts in the size and frequency of contacts involving the target, measuring before and after the event
- **response time**—look at the overall response time for service requests (the average interval between request and response) into the network, then determine if it has increased during and following the event
- **contact rate**—look at the relative rate of contact with services (indicated by port and protocol) on the targeted network, searching for shifts in the contact rate and the size of traffic on those services
- **hosts**—look at the set of hosts in contact with the target, and determine if it has shifted unusually during and after the event.

This section will explore the first of these alternatives: looking at traffic shifts. (The other alternatives may be useful to analysts, but are not covered here.)

3.1.1 Examining Shifts in Traffic

Sofia is a cybersecurity analyst who's been asked to investigate changes in network traffic that occurred before, during, and after a suspected incident. Her investigation applies the Formulate-Model-Analyze steps in the SiLK workflow (described in Section 1.5) to perform a single-path analysis that looks at shifts in network traffic around the event. Her results will help to guide her team's investigation of and response to the incident.

Sofia starts by knowing a tentative start and end time for the event (14:00-1500 on June 17, 2015), and that the event involves covert data transfer. First, she will filter for network flow records associated with the targeted host around the time of the event (the Formulate step). This involves pulling records from the appropriate parts of the repository and isolating those that involve the targeted host. Next, she'll divide the set of records into bins according to volume and compute counts for each bin before, during, and after the event (the Model step). Finally, she will interpret the counts to assess the potential impact of the event (the Analyze step).

These steps are described in more detail below.

Filter Traffic Around the Event (Formulate)

Sofia structures the Formulate portion of her analysis as a query with the `rwfilter` tool. She knows when the incident occurred and uses this information to determine which parts of the repository to query by date-hour (using the parameters `--start` and `--end`) and type (using the `--type` parameter). The association with the target host and sensor are indicated by the host's IP address (using the `--any-address` parameter) and sensor name (using the `--sensor` parameter). The filtered records are then stored in a file (using the `--pass` parameter) for later parts of the analysis.

Summarize Records (Model)

Next, Sofia summarizes the results of her initial query as part of the Model step in her analysis. After she pulls the flow records via her query, she uses the `rwuniq` command to summarize them. Her goal is to filter out the appropriate group of flows to summarize by volume. She will therefore create volume-based groups at low, medium, and high values for both byte volume and flow duration.

For each group, Sofia uses another call to `rwfilter` to pull records from the file generated previously. It extracts those with the volume measure for each group (using either `--bytes` or `--duration`). The output goes to `rwuniq` to count the records. Her analytic generates separate counts for each hour and type of flow record using the options `--fields=stime,type` and `--bin-time=3600`. The number of records in each group are counted using `--values=records`.

Interpret Counts (Analyze)

Sofia's investigation provides a high-level view of the variation in activity from an hour before the event to an hour after it. She can now analyze these results to see how network activity varied during these time periods. This will give her a better idea of the incident's impact.

3.1.2 How to Profile Traffic

The resulting set of commands for Sofia's analysis are shown in Example 3.1.

1. Sofia's initial query is shown in Command 1. It calls the `rwfilter` command to retrieve records from the flow repository that were stored during the two hours surrounding the event (`--start=2015/06/17T13` and `--end=2015/06/17T15`). It filters these records by type (`--type=in,inweb,out,outweb`) and whether they show communication with the target host (`--any-address=192.168.70.1`, which filters flow records that have this IP address as either a source or a destination). It then saves the resulting records to the file `traffic.rw` in the local directory. This file serves as the basis for profiling traffic around the event.

2. Her next step is to summarize the flow records by volume. Command 2 calls `rwfilter` to find low-volume flows in `traffic.rw`. It filters out flows with low byte counts (`--bytes=0-300`) and sends them to standard output (`--pass=stdout`), then uses a pipe (`|`) to direct these records to the `rwuniq` command. The `rwuniq` command divides the records into hour long (i.e., 3600 second long) bins (`--bin-time=3600`) and counts the records in each bin according to their start time range and type (`--fields=stime,type`). It then stores the resulting hourly counts in the file `low-byte.txt`.

3. In Commands 3 and 4, she similarly uses `rwfilter` and `rwuniq` to generate hourly counts of medium and high-volume flows. Command 3 counts records for medium-volume flows (`--bytes=310-100000`) and stores the results in the file `med-byte.txt`. Command 4 counts records for high-volume flows (`--bytes=100001-`, which does not specify an upper limit on byte counts) and stores the results in the file `high-byte.txt`.

4. Sofia then uses `rwfilter` and `rwuniq` to summarize flows in `traffic.rw` by duration, as shown in Command 5. It first calls `rwfilter` to filter flows that are up to a minute long (`--duration=0-60`). As in Commands 2-4, it then calls `rwuniq` to divide these records into hour-long bins and count the flows in each bin by start time and type. Finally, it stores the resulting hourly counts in the file `short-duration.txt`.

5. In Commands 6 and 7, she again calls `rwfilter` and `rwuniq` to generate hourly counts of medium and long-duration flows. Command 6 counts medium-duration flows (`--duration=61-120`) and stores the results in the file `med-duration.txt`. Command 7 counts long-duration flows (`--duration=121-`) and stores the results in the file `long-duration.txt`.

```
<1>$ rwfilter --start=2015/06/17T13 --end=2015/06/17T15 \
   --sensor=S1 --type=in,inweb,out,outweb \
   --any-address=192.168.70.10 --pass=traffic.rw
<2>$ rwfilter traffic.rw --bytes=0-300 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >low-byte.txt
<3>$ rwfilter traffic.rw --bytes=301-100000 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >med-byte.txt
<4>$ rwfilter traffic.rw --bytes=100001- --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >high-byte.txt
<5>$ rwfilter traffic.rw --duration=0-60 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >short-duration.txt
<6>$ rwfilter traffic.rw --duration=61-120 --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >med-duration.txt
<7>$ rwfilter traffic.rw --duration=121- --pass=stdout \
| rwuniq --bin-time=3600 --fields=stime,type \
   --values=records --sort-output >long-duration.txt
```

Example 3.1: Using `rwfilter` and `rwuniq` to Profile Traffic Around an Event

The results of these commands are collated in Example 3.2. The counts show that there was a marked increase in low-to-medium byte and short-to-medium duration web traffic during and after the event. There was no corresponding increase in high byte or long duration traffic.

Based on these results, Sofia can start to focus her investigation on what common factors exist in the increased traffic. Her goal is to build awareness of the impact of the event in a way that helps responders to deal with that impact.

stime	type	sbyte	mbyte	hbyte	sdur	mdur	ldur
2015/06/17T13:00:00	in	720	160		880		
2015/06/17T13:00:00	inweb	5	352		357		
2015/06/17T13:00:00	out	764	192	9	960	5	
2015/06/17T13:00:00	outweb	1	400		401		
2015/06/17T14:00:00	in	1449	66		1515		
2015/06/17T14:00:00	inweb	12	346		358		
2015/06/17T14:00:00	out	1500	96	3	1595	1	1
2015/06/17T14:00:00	outweb	8339	10051	2	18382	9	1
2015/06/17T15:00:00	in	2528	550		3077	1	
2015/06/17T15:00:00	inweb	14	345		359		
2015/06/17T15:00:00	out	2520	558	5	3076	3	3
2015/06/17T15:00:00	outweb	10309	11072	7	21366	12	9

Example 3.2: Collated Profile of Traffic Around an Event

3.2 Generate Top *N* Lists

Filtering flow records by time, sensor, type, and volume characteristics often produces groups that contain flow records of interest. However, these groups also contain extraneous flows that produce noise, which makes it more difficult to spot patterns in the data.

One strategy for removing these extraneous flows is to identify the largest sub-groups, validate each sub-group, and either set it aside or include it in the collection of flows of interest. The sub-groups are identified by a combination of flow characteristics, such as the IP address of the source, the TCP flags present in the flow, or the network service involved. The contribution of each sub-group is measured by the total bytes or packets per sub-group, the number of records per sub-group, the number of distinct values present for some field in the flow record, or another summary statistic.

The overall process of pulling a collection of flow records and then removing flows not related to the analysis is sometimes referred to as *top-down analysis*. There is also a *bottom-up analysis* that involves starting with a minimal set of records that are of interest, then basing further queries to isolate more records of interest based on the field values in the minimal set.

Our analyst Sofia is continuing the investigation she began in Section 3.1. Her next task is to look into host activity before, during, and after the event. She will again apply the Formulate-Model-Analyze workflow described in Section 1.5 to perform a top-down analysis and generate top-N lists of hosts that exhibit different behaviors of interest.

3.2.1 Using `rwstats` to Create Top *N* Lists

Sofia uses the `rwstats` command to find the identity and relative size of the sub-groups of interest. This command explicitly includes parameters to limit output to the largest contributors and describes the contribution

of those categories to the overall flow collection[9].

Without `rwstats`, Sofia could load flow records into a spreadsheet and then generate a pivot table to identify the most common characteristics. `rwstats` is much faster and easier to use than a spreadsheet. It deals with high numbers of flow records very efficiently in terms of storage and memory usage.

Removing Unwanted Flows and Finding Hosts of Interest (Formulate)

When generating top-N lists, the Formulate stage involves eliminating flow records for network activity that is not of interest. This activity, sometimes referred to as *network chaff*, may include connections with not enough data exchanged to be significant, those involving services that are not important for an analysis, or, in general, anything that could obfuscate the results by contaminating the flow records retrieved for the event.

Sofia will filter the results from her earlier investigation to remove network chaff and find hosts with traffic that's relevant to her current investigation. Her analysis is shown in Commands 1 through 4 of Example 3.3.

1. In Command 1, Sofia looks at the first few flows in the `traffic.rw` file generated in Example 3.1. The sequence of flows shown are DNS queries (i.e., flows from port 53). There is not enough information at the flow level to indicate whether they are relevant to the event that she is investigating.

2. Command 2 uses a new call to `rwfilter` to exclude the unwanted DNS flows from `traffic.rw` and sends the output to `rwcut` for further examination. The results show that the host at `192.168.70.10` is doing a lot of communication on TCP port 8082, which is associated with a file management utility known as Utilistore. The larger-volume flows appear to be associated with the host at `10.0.40.83`.

3. Command 3 queries the flow repository to look for flows showing communication on port 8082 with at least 150 average bytes per packet across the full data set. The results of this query are then passed to `rwuniq` to profile all of the locations to which data has been sent. The results of this profile show three hosts receiving this traffic, including both `192.168.70.10` and another host at `192.168.200.10`.

4. In Command 4, Sofia makes a further call to `rwfilter` to pull all of the traffic associated with the newly located addresses `10.0.40.83` and `192.168.70.10`. These addresses appear twice in the `rwfilter` call to specify that both the source and destination are constrained to these addresses. The results are then stored in the file `traffic2.rw` to be examined further.

Hint 3.1: CIDR Notation for IP Addresses

Note that the addresses in Command 4 of Example 3.3 are specified with `--scidr` (Source CIDR block) and `--dcidr` (Destination CIDR block) instead of `--saddress` (source address), `--daddress` (destination address), or `--any-address` (both source and destination addresses). The `--scidr` and `--dcidr` parameters accept comma-separated lists of addresses in CIDR notation, whereas the other parameters accept only a single address. The /32 CIDR notation specifies a single address and thus permits us to use a list of addresses.

[9] `rwstats` has further functions that facilitate describing collections of flows statistically; see the `man` page for `rwstats`.

```
<1>$ rwcut --fields=1-3,protocol,bytes --num-recs=5 traffic.rw
            sIP|            dIP|sPort|pro|     bytes|
     10.0.40.20|  192.168.70.10|   53| 17|       242|
     10.0.40.20|  192.168.70.10|   53| 17|       242|
     10.0.40.20|  192.168.70.10|   53| 17|       242|
     10.0.40.20|  192.168.70.10|   53| 17|       242|
     10.0.40.20|  192.168.70.10|   53| 17|       242|
<2>$ rwfilter traffic.rw --aport=0,53 --fail=stdout \
| rwcut --fields=1-5,bytes --num-rec=5
            sIP|            dIP|sPort|dPort|pro|     bytes|
     10.0.40.27|  192.168.70.10|44358| 8082|  6|       332|
     10.0.40.27|  192.168.70.10|44383| 8082|  6|       332|
     10.0.40.83|  192.168.70.10|53596| 8082|  6|       838|
     10.0.40.83|  192.168.70.10|53597| 8082|  6|       551|
     10.0.40.83|  192.168.70.10|53598| 8082|  6|      1080|
<3>$ rwfilter --start=2015/06/13 --end=2015/06/18 --type=all \
   --proto=6 --dport=8082 --bytes-per=150- --pass=stdout \
| rwuniq --fields=dip --values=flows,distinct:bytes
            dIP|   Records|bytes-Dist|
      155.6.3.1|         1|         1|
 192.168.200.10|       804|        32|
  192.168.70.10|      1132|        37|
<4>$ rwfilter --start=2015/06/13 --end=2015/06/18 \
   --type=all --scidr=10.0.40.83/32,192.168.200.10/32 \
   --dcidr=10.0.40.83/32,192.168.200.10/32 \
   --pass=traffic2.rw
<5>$ rwstats --fields=dip,dport --values=flows,bytes --count=6 \
   traffic2.rw
INPUT: 11846 Records for 1954 Bins and 11846 Total Records
OUTPUT: Top 6 Bins by Records
            dIP|dPort|   Records|     Bytes|  %Records|   cumul_%|
 192.168.200.10| 8082|      5906|   8252849| 49.856492| 49.856492|
     10.0.40.83|56018|        45|      6357|  0.379875| 50.236367|
     10.0.40.83|55026|        18|      4653|  0.151950| 50.388317|
 192.168.200.10|  137|        15|      3510|  0.126625| 50.514942|
     10.0.40.83|  771|        15|      2520|  0.126625| 50.641567|
     10.0.40.83|56348|         3|      3390|  0.025325| 50.666892|
<6>$ rwstats --fields=dip,dport --values=bytes,flows --count=6 \
   traffic2.rw
INPUT: 11846 Records for 1954 Bins and 39548165 Total Bytes
OUTPUT: Top 6 Bins by Bytes
            dIP|dPort|     Bytes|   Records|    %Bytes|   cumul_%|
     10.0.40.83|49375|  13139355|         3| 33.223678| 33.223678|
 192.168.200.10| 8082|   8252849|      5906| 20.867843| 54.091521|
     10.0.40.83|54964|   1488312|         3|  3.763290| 57.854811|
     10.0.40.83|49408|   1488312|         3|  3.763290| 61.618100|
     10.0.40.83|54345|    328470|         3|  0.830557| 62.448657|
     10.0.40.83|54404|    328470|         3|  0.830557| 63.279214|
```

Example 3.3: Removing Unneeded Flows for Top N

Summarizing Destination Port Usage By Records and Bytes (Model)

After Sofia queries and filters the flow records to isolate those of interest, she can calculate values to clarify her understanding of these data (the Model step). She could use either `rwuniq` or `rwstats` to understand the contributors to these data, which fed into the filtering process. `rwstats` allows for more explicit limits on the number of bins that are displayed. In contrast, `rwuniq` shows all of the bins for the input dataset. `rwstats` also shows the percentage contribution to the overall input of each bin and cumulatively across bins. These limits are often expressed as a count of bins, but they can also be expressed in terms of percentage contribution or a threshold on the count. The percentages are calculated based only on the first value specified for the bin. This allows `rwstats` to be used flexibly to profile the contributors to the data.

Sofia's investigation into the behavior of the hosts she found earlier is shown in Commands 5 and 6 of Example 3.3.

1. Command 5 uses `rwstats` to profile the records in `traffic2.rw`. It looks at the destination port utilization in these data by the flow count (as a rough measure for how often communication takes place), with bytes also calculated as a supplementary value.

   In the results, the largest contributor accounts for very close to half of the data. This is not surprising, since she used this port to identify these hosts as being of interest during the filtering process. Three of the other ports shown are ephemeral ports (officially, ports numbering 49,152 or more, although some Linux versions use 32,768 to 61,000, and some Windows versions use 1,025-5,000). This port usage indicates that `192.158.200.10` is the server and `10.0.40.83` is the client. These two IP addresses account for at most a little over a third of one percent of the records. Port 137 is the Windows netbios name service port; port 771 is an ICMP data artifact that will be discussed below.

2. For a contrasting look at the data, Command 6 calls `rwstats` to summarize port utilization by the number of bytes. This serves as a rough measure of the size of the communication taking place. By this measure, the largest contributor is not the traffic on port 8082, but rather traffic on an ephemeral port. This shows the need for analysts to examine the data from several perspectives to clarify its interpretation.

3.2.2 Interpreting the Top-N Lists

Sofia can now interpret the top-N lists from Example 3.3 (the Analyze step in the SiLK workflow). One key to interpreting these results is provided in the output from Command 3. The last column of results is the count of distinct values for the bytes in each flow that is assigned to that bin. In this case, it is the number of bytes in each flow going to a specific IP address. For the last two entries, the value is less than 10 percent of the record count, indicating that communication with the same number of bytes is common. This in turn suggests that this traffic is automated rather than human-driven.

In this light, Sofia interprets the results shown from Command 5 as suggesting that this traffic averages about 1,400 bytes per flow to the server, with smaller acknowledgement traffic being returned to the client via the ephemeral ports. This suggestion, however, is contradicted by the results shown from Command 6, which show several very large-byte flows occur to the clients on the ephemeral port. This indicates to Sofia that the data transfer is bi-directional.

Confirming the data transfer dynamics and determining if any indications of threat are present requires further analysis—pulling more traffic to see if these hosts shift behavior across time as threats in their contacts to additional hosts.

When Sofia looks at the results for Command 5, she sees that the traffic to UDP port 137 (name service) is not answered with service traffic. Instead, the response involves messages using protocol 1, ICMPv4 (which appears with a 771 port number, although ICMP does not use ports). This is suggested by the common number of flows associated with these ports. This interpretation was confirmed by an inspection of the data using `rwcut` that was too long to show in the example. The flow generators encode the ICMP message type and code in the dPort field of NetFlow and IPFIX records.

Sofia knows that the value of 771 corresponds to a message indicating that name service is unreachable. While the number of repetitions is not extensive (15 across 3 days of traffic), that repetition despite unreachable service indicates that the server is generating the port 137 traffic automatically.

This page intentionally left blank.

Chapter 4

Intermediate Multi-path Analysis with SiLK: Explaining and Investigating

This chapter introduces intermediate multi-path analysis through application of the analytic development process with the SiLK tool suite. It discusses iteration, conditional analysis steps, categorization, and behavior identification.

Upon completion of this chapter you will be able to

- describe intermediate multi-path analysis and how it maps to the analytic development process
- describe SiLK tools commonly used with intermediate multi-path analysis
- provide example multi-path network flow analysis workflows
- describe how to apply the multi-path analysis workflow to situational awareness

4.1 Multi-path Analysis: Concepts

4.1.1 What Is Multi-path Analysis?

Some network behaviors are not visible within the single view of the network flow data provided by single-path analysis. Finding them requires investigating and integrating several different views of the data. This type of analysis is known as *multi-path analysis*.

A multi-path analysis involves a deeper, multi-pronged dive into network flow data than can be accomplished with a single-path analysis. While a single-path analysis may involve looking at summary data or just one part of a data set, multi-path analysis explores different aspects of the data set (ports, IP addresses, protocols, packet and byte volumes, flow types and volumes, etc.) to find trends, leftovers, and groupings that are not necessarily visible in a single view of the data. Multi-path analysis builds upon single-path analysis; often, a single-path analysis is performed as just one phase of a multi-path analysis.

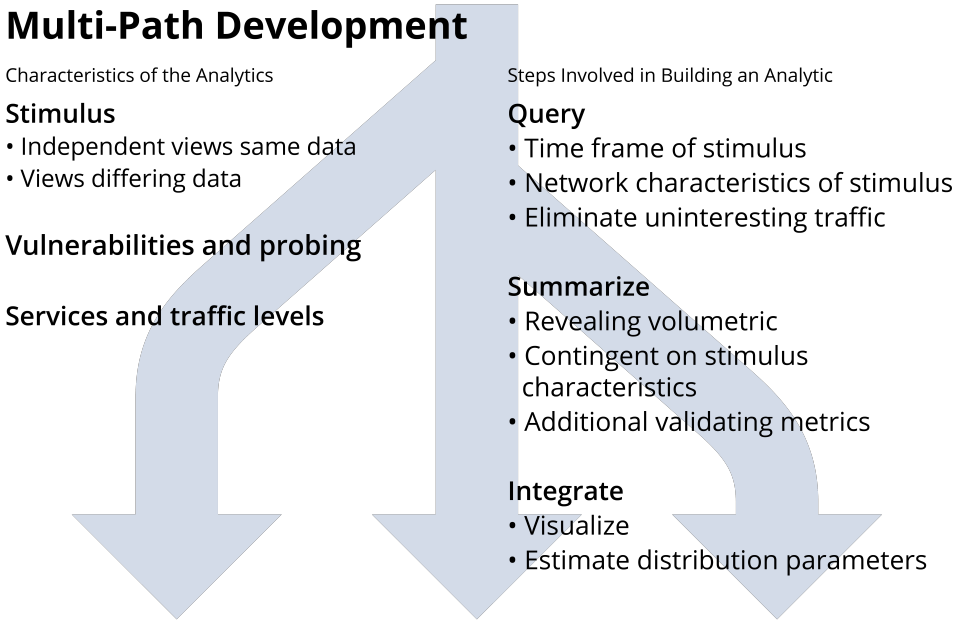

Figure 4.1: Multi-Path Analysis

Multi-path analysis follows the general analysis framework described in Section 1.5. The overall process includes several steps:

1. Formulate the problem and gather context about the data. Is this event similar to other incidents? Which aspects of the data set should be investigated? How should we classify and group the data?

2. Model and test the data. What network behaviors are we looking for? Which statistics and metrics give us insight into these behaviors of interest?

3. Analyze the results. What did we find? How can we integrate the results of our different investigations into the data? Is our model of the event borne out by the results of our analysis?

4. Refine the analysis. Now that we have an idea of what might be going on, we can optionally take another look at the data and our assumptions to improve the analysis.

In multi-path analysis, the Formulate and Model steps are performed on multiple categories of data, each associated with aspects of the overall behavior of interest. After gathering context about the data, retrieving the data from the SiLK repository, building a model, and summarizing the data, a multi-path analysis includes an integration and analysis step that ties together the separate results to further characterize network behavior. The analyst can then iterate these steps to further refine the analysis.

4.1.2 Example of a Multi-path Analysis: Examining Web Service Traffic

A short example of a multi-path analysis is shown in Example 4.1. Chris is a network administrator who wants to investigate outbound web traffic. Their analysis will look at statistics for outbound web traffic on ports that typically carry this type of traffic.

As a network administrator, Chris is interested in validating the regularity of service provided by the network's web servers, and in understanding what demands are being made by web clients. They've decided to look at a week's network traffic, between June 15 and June 21, 2015.

During the Formulate step of the multi-path analysis, Chris gathers information about related categories of data. Multiple ports are used for web traffic: the normal HTTP port (TCP port 80), the HTTP secure port (TCP port 443), the alternate web ports (TCP ports 8080 and 8443), and so forth. Chris would like to examine outbound traffic on each of these ports.

A single-path analysis would gather flows from all of these ports into one pool of data to produce a composite model in the next step. In contrast, the multi-path analysis in Example 4.1 gathers traffic from each port as a separate pool of data. Chris will investigate each pool of port traffic individually during the Model step, profiling individual characteristics and summarizing variations.

1. In Command 1, Chris uses the `mkfifo` command to create a set of named pipes, or FIFO (first-in-first-out) files, to support a complex series of `rwfilter` calls in Command 2. Each one is named after a port number (e.g., `multi-port8080.fifo`) to indicate that it carries data related to that port.

Hint 4.1: Use Named Pipes for Efficient Analytics

Named pipes efficiently transfer data from one SiLK command to another without the overhead of writing data to a disk file at each step, making the analytic run more quickly. Using named pipes also makes the analytic easier to script. Unlike regular operating system pipes (`|`), named pipes persist after a command finishes executing and can therefore be used as a source of input data for subsequent SiLK commands.

See Chapter 9 for additional tips on how to improve the performance of your SiLK analytics.

2. In Commands 2 through 5, Chris sets up an analytical structure known as a *manifold*, which is discussed in more detail in section 4.2.1. The pipelined series of `rwfilter` calls in Command 2 first pulls from the repository all outbound complete TCP flows (shown in the selection and partitioning commands in the first `rwfilter` call). These outbound flows (selected by `--type=out,outweb`) include traffic produced by the monitored network. Selecting complete TCP flows (partitioned by a combination of `--proto`, `--flags-all`, `--packets`, and `--bytes-per`) avoids data that includes scans, extended sessions, redundant flow termination, and other data artifacts—all of which might confuse the analysis.

3. The subsequent `rwfilter` calls in Command 2 then divide the retrieved flow records into separate pools (one for each of the web-related ports) using both standard output and the FIFO files. Each of these `rwfilter` calls culminates (at the end of Command 2 and in Commands 3 through 5) in a call

```
<1>$ mkfifo /tmp/multi-port8080.fifo; \
     mkfifo /tmp/multi-port443.fifo; \
     mkfifo /tmp/multi-port8443.fifo
<2>$ rwfilter --start=2015/06/15 --end=2015/06/21 \
   --type=out,outweb --proto=6 --flags-all=SAF/SAF,SAR/SAR \
   --packets=4- --bytes-per=65- --pass=stdout \
| rwfilter stdin --aport=8080 \
   --pass=/tmp/multi-port8080.fifo --fail=stdout \
| rwfilter stdin --aport=443 --pass=/tmp/multi-port443.fifo \
   --fail=stdout \
| rwfilter stdin --aport=8443 \
   --pass=/tmp/multi-port8443.fifo --fail=stdout \
| rwfilter stdin --aport=80 --pass=stdout \
| rwbag --bag-file=sipv4,bytes,./output/tmp80.bag &
<3>$ rwbag --bag-file=sipv4,bytes,./output/tmp443.bag \
   /tmp/multi-port443.fifo &
<4>$ rwbag --bag-file=sipv4,bytes,./output/tmp8443.bag \
   /tmp/multi-port8443.fifo &
<5>$ rwbag --bag-file=sipv4,bytes,./output/tmp8080.bag \
   /tmp/multi-port8080.fifo &
<6>$ wait
<7>$ rwbagcat --print-stat=./output/out80.txt \
   ./output/tmp80.bag
<8>$ rwbagcat --print-stat=./output/out8080.txt \
   ./output/tmp8080.bag
<9>$ rwbagcat --print-stat=./output/out443.txt \
   ./output/tmp443.bag
<10>$ rwbagcat --print-stat=./output/out8443.txt \
   ./output/tmp8443.bag
<11>$ grep 'mean' ./output/out{80,8080,443,8443}.txt
./output/out80.txt:              mean:  5.471e+06
./output/out8080.txt:              mean:  4.707e+07
./output/out443.txt:              mean:  2.585e+07
./output/out8443.txt:              mean:  1.367e+08
<12>$ grep 'standard' ./output/out{80,8080,443,8443}.txt
./output/out80.txt:standard deviation:  2.021e+07
./output/out8080.txt:standard deviation:  6.655e+07
./output/out443.txt:standard deviation:  5.704e+07
./output/out8443.txt:standard deviation:  9.266e+06
```

Example 4.1: Examining Flows for Web Service Ports

to `rwbag` to generate a summary set of byte counts per source IP address in each pool. Section 4.2.4 describes more about the SiLK bag tools.

4. To make all of this work in Linux, these commands have to operate in a producer-consumer manner as background processes. Command 6 is the shell command `wait`, which causes the analytic to wait until the producer-consumer processes have finished. This ensures that that the counts are all complete.

5. Commands 7 through 10 use `rwbagcat` to calculate descriptive statistics for each of the pools of data that were stored in bags during Commands 2 through 5. They send each set of statistics to a separate text file.

6. Chris can now inspect the summary statistics for outbound web traffic on ports 80, 443, 8080 and 8443 to see if unusual amounts of traffic are going through these ports. Commands 11 and 12 show two of these statistics, mean and standard deviation, across the pools of data to produce integrated summaries of outbound web traffic on these ports.

4.1.3 Exploring Relationships and Behaviors With Multi-path Analysis

Many possible relationships in network traffic can be explored during multi-path analyses, in addition to the port-protocol alternatives in the web traffic analysis shown in Example 4.1.

- **Address relationships** can be explored by looking at the behavior of a block of addresses as seen by varying sensors, looking at variations in behavior between addresses within a block, or looking at behaviors of several blocks within a given Autonomous System that is handled by a common route. These address relationships might be useful for analysis tasks such as confirming suspected malware propagation or isolating targeted scanning from more general scanning.

- **Timing relationships** can be explored by separately summarizing and examining behavior before, during, and after the times associated with network events. They can also be explored by profiling behavior around an event in comparison to a period of normal activity that goes on for a similar period of time. These timing relationships could be useful to identify more subtle intrusions, for example, or to isolate activity that might indicate malicious pivoting between parts of the network.

- **Volumetric relationships** can be explored to find more complex relationships between pools of data depending on byte volumes or transfer rates. These volumetric relationships could help to indicate covert data exfiltration, for example, or detect when services are exploited to support malicious activity.

4.1.4 Integrating and Interpreting the Results of Multi-path Analysis

During the Model phase of the analysis, the focus is often about determining values to provide insight on the relationships in the data. Measures of central tendency (such as averages) are frequently useful. They should be extended by measures of extent (such as range or standard deviation) to provide context for variation between pools of data. With these measures calculated, a range of activity can be specified. In the web traffic analysis shown in Example 4.1, for instance, each pool of network traffic associated with a port is profiled into an output text file using the `rwbagcat --print-stat` command, which computes a variety of descriptive statistical measures, including mean and standard deviation.

Once the measures are calculated, integrate them by matching them across the pools of data to establish trends or contrast values for the identified relationships. In Example 4.1, the standard deviations are almost

all larger than the mean values, indicating that the count distribution has a long tail. More traffic values are lower than the mean than higher, but the higher ones go quite high.

Interpreting the results involves examining the statistics given and applying what insight the analyst can provide. Consider a variety of explanations for the behavior. If necessary, refine and iterate the analysis to support or deny these explanations.

Generally, consider benign interpretations for behavior first, and only reject them if appropriate contradiction can be found. In Example 4.1, the benign interpretation is that much of the traffic across these ports cannot be readily differentiated; the main difference is which port it was sent on. An alternative interpretation is that the large bulk of relatively low-byte traffic and the long tail of relatively high-byte traffic should be examined separately to determine if suspicious traffic is present. This would involve iterating the analysis to partition and separately examine low-byte and high-byte traffic for each port.

4.1.5 "Gotchas" for Multi-path Analysis

While it is quite possible to explore combinations of relationships within a given analysis, some care is needed. As the number of relationships being combined increases, so does both the size and the complexity of the results. This raises several concerns in performing combined multi-path analyses:

- Interpreting the results can require a lot of rather tedious effort for limited results. It is likely that the number of data combinations that need to be evaluated will yield many cases that are not of interest to the investigation, but may still need to be explored either for completeness or to be sure that all interesting cases are dealt with. If the combination of data relationships is not carefully chosen, analysts may spend a lot of time without much compensating insight.
- The amount of data required to fully explore combinations of relationships can be extensive, which will both slow the gathering step and require iteration across cases. Finding appropriate data to cover combinations can involve effort—for example, establishing that "normal" network activity does not itself contain malicious activity!
- As the number of combinations involved in analysis increases, so does the possibility that observed differences may occur by coincidence. This can lead to unintended "cooking" of the results, focusing on combinations that confirm the analysts' preconceptions, rather than on a more holistic view of the behaviors.

Based on these concerns, we recommend that you **keep your multi-path analyses as simple as possible**, given the behaviors being studied. It may well be preferable to perform a series of simpler analyses rather than a single, complex, multi-factor analysis—both more manageable in action and producing more easily understood results.

4.2 Multi-path Analysis: Analytics

The SiLK commands, parameters, and examples described in this chapter can be employed with any analysis method. However, they involve more complex uses of the SiLK tool suite than the commands described in Chapter 2. Multi-path analyses frequently make use of these commands to construct analytics and store intermediate and final results.

4.2.1 Complex Filtering With `rwfilter`

Complex filtering combines operating system pipes with `rwfilter` output parameters to perform large-scale, multi-path flow analyses. It is an important concept to adopt and incorporate into the multi-path analysis process discussed in Section 4.1 and the overall SiLK workflow described in Section 1.5.

The conditional steps, refinement, and iteration of multi-path analysis usually require multiple `rwfilter` queries. Wide time periods, large network ranges, and increasing network traffic are a few examples that complicate this process. Unfortunately, the increasing `rwfilter` directory and file searches needed to support such analyses also impact disk input/output (I/O) performance and latency.

To address these trade-offs, `rwfilter` provides three output parameters to optimize repository queries and classify traffic behavior.

- `--pass-destination` writes flow records that pass *all* partitioning criteria to a path.
- `--fail-destination` writes flow records that fail *any* partitioning criteria to an alternate path.
- `--all-destination` writes *all partitioned flow records* to a third path.

These parameters are so commonly used that the remainder of this handbook will refer to them by their common abbreviations (`--pass`, `--fail`, and `--all`). All three can be combined and repeated within the same `rwfilter` statement. They can write network flow records to a newly-created file, a named pipe, standard output (`stdout`), or standard error (`stderr`).

Multi-level Filtering With Pipes and Manifolds

A *manifold* combines several `rwfilter` commands to categorize traffic. By using operating system pipes and switches with the `--pass` and `--fail` parameters, analysis can chain multiple `rwfilter` statements together to reduce an initial broad data pull into smaller sets of results that isolate traffic of interest for further analysis. After examining your manifold's initial categorization of network flow data, you can adjust the parameters of subsequent `rwfilter` calls to re-categorize data and find additional records of interest.

Chaining `rwfilter` calls into a manifold also can reduce the number of files saved to disk, since the output from each `rwfilter` call is passed to the next one via UNIX pipes. This can improve the analytic's performance.

Manifolds perform complex traffic categorization by filtering data into overlapping and non-overlapping traffic categories.

Manifolds with Non-overlapping Traffic. If the categories are *non-overlapping*, the manifold uses the `--pass` parameter to write matching traffic to a file, as shown in Figure 4.2. The shaded arrows indicate data flows from the SiLK repository to the series of `rwfilter` commands that comprise the manifold. Each `rwfilter` command uses the `--pass` parameter to save the records that meet the filtering criteria to a file. It also uses the `--fail` parameter to transfer the remaining traffic to the next `rwfilter` command for further filtering. The final `rwfilter` command in Figure 4.2 saves the last category and uses the `--fail` parameter to discard the remaining uncategorized traffic. (Alternatively, the manifold could save the discarded traffic to a file.)

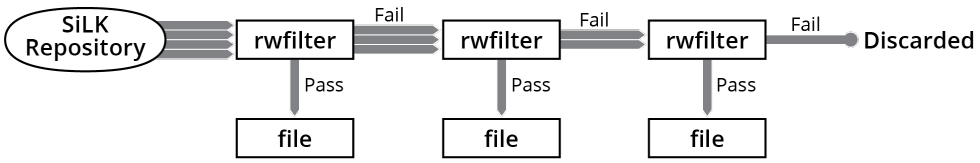

Figure 4.2: Diagram of a Simple, Non-overlapping Manifold

Manifolds with Overlapping Traffic. If the categories are *overlapping*, the manifold again uses the `--pass` parameter to filter traffic in a category. But it would use the `--all` parameter as shown in Figure 4.3 to transfer *all* traffic to the next call to `rwfilter`. With overlapping categories, the manifold is not done with a record just because it assigned a first category to it. The record may belong to other categories as well, which would be identified by subsequent calls to `rwfilter`.

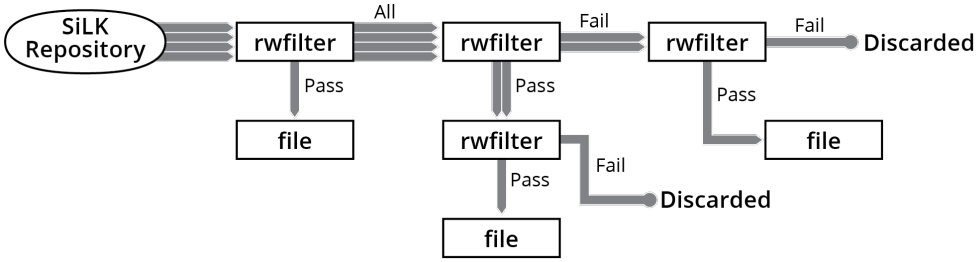

Figure 4.3: Diagram of a Complex, Overlapping Manifold

Figure 4.3 also shows how records can be passed to subsequent `rwfilter` calls via the `--fail` parameter. Using successive `rwfilter` calls that combine the `--pass`, `--fail`, and `--all` parameters gives you a high degree of control over how your network traffic is categorized for analysis.

Simple Manifold: Filtering Incoming Client and Server Traffic

In Example 4.2, we look for inbound flows from external servers and clients. The manifold sorts these flows into two separate files. This is an example of a non-overlapping manifold, since incoming client and server traffic have independent characteristics.

```
<1>$ rwfilter --start=2015/06/17T15 --sensor=S5 --type=in \
   --protocol=6 --flags-initial=SA/SA --packets=3- \
   --fail=stdout --pass=inbound-clients.rw \
| rwfilter stdin --flags-initial=S/SA --packets=3- \
   --pass=inbound-servers.rw
```

Example 4.2: Simple Manifold to Select Inbound Client and Server Flows

- The first `rwfilter` command in the manifold filters out all incoming client traffic.
  - The `--start` and `--sensor` selection parameters specify the time period and network sensor of interest.

  - `--protocol=6` selects Transmission Control Protocol (TCP) traffic.
  - `--flags-initial=SA/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN/ACK", indicating that the source IP address is a server.
  - `--packets=3-` selects flow records with three or more packets to identify TCP sockets.
  - `--pass=incoming-client.rw` stores flow records that match the filtering criteria to the file `incoming-client.rw`.
  - `--fail=stdout` sends all records that do not match the filtering criteria to the second part of the manifold.

- The second `rwfilter` command in the manifold selects incoming server traffic.
  - `stdin` tells the `rwfilter` command to process flow records from standard input (i.e., the output from the previous `rwfilter` command).
  - The command does not filter on Sensor, Type and Protocol. It looks at traffic that does not meet those criteria.
  - `--flags-initial=S/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN" with no "ACK," indicating the source IP address is a client.
  - `--packets=3-` selects flow records with three or more packets, which is the minimum number required to open a TCP socket.
  - `--pass=incoming-server.rw` stores flow records that match the filtering criteria to the file `incoming-server.rw`
  - Notice that the second `rwfilter` command did not specify a `--fail` destination. We are only interested in the records that pass the filtering criteria in the second part of the manifold, so there is no need to save the ones that fail.

Hint 4.2: An Example TCP Session

Figure 4.4 shows the TCP flags for client and server communication. The client first sends a packet with the SYN flag to initiate the TCP connection. The server responds with a packet with SYN and ACK flags to acknowledge the client request. The client then sends a packet with the ACK flag to acknowledge receipt of the server SYN/ACK packet; it follows with one or more packets that have PSH and ACK flags. The server responds to each client PSH/ACK packet with packets that also have PSH and ACK flags. At this stage of client/server communication, they have established a TCP socket and exchanged data.

To complete the session, the client sends the server a packet with FIN and ACK flags. The server responds to the client with a packet that has an ACK flag to acknowledge receipt of the client's FIN/ACK packet. The server follows with a packet that has FIN and ACK flags. The client then responds by sending a packet with a ACK flag, responding to the server's FIN/ACK packet, ending the session. For more detailed information about TCP flags, see Appendix A.2.

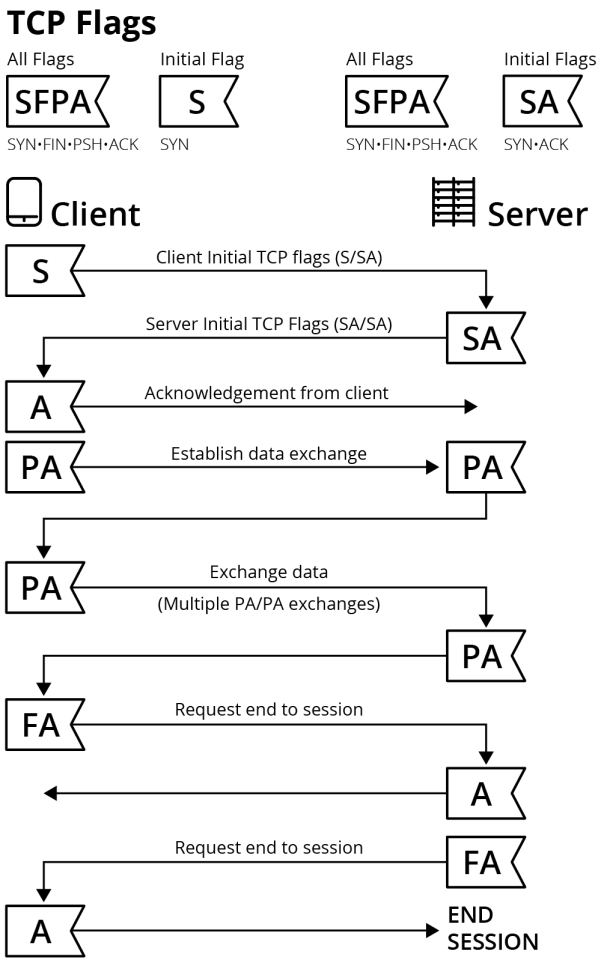

Figure 4.4: Client and Server TCP flags

Expanding the Simple Manifold: Filtering for Incoming and Outgoing Client and Server Traffic

To expand the simple manifold in the previous example to partition both inbound and outbound traffic, we make three modifications as shown in Example 4.3.

1. Prefilter traffic to ignore flows that we know are unwanted. These consist of non-TCP flow types other than `in` and `out` plus flows with fewer than three packets.
2. Filter traffic from internal and outgoing servers and clients into files.
3. Store the leftover flows in a file for later use.

```
<1>$ rwfilter --start=2015/06/17T15 --sensor=S5 --type=in,out \
   --protocol=6 --packets=3- --pass=stdout \
| rwfilter stdin --type=in --flags-initial=SA/SA \
   --pass=incoming-server.raw --fail=stdout \
| rwfilter stdin --type=in --flags-initial=S/SA \
   --pass=incoming-client.raw --fail=stdout \
| rwfilter stdin --type=out --flags-initial=SA/SA \
   --pass=outgoing-server.raw --fail=stdout \
| rwfilter stdin --type=out --flags-initial=S/SA \
   --pass=outgoing-client.raw --fail=leftover.raw
<2>$ for f in incoming-server.raw incoming-client.raw \
   outgoing-server.raw outgoing-client.raw leftover.raw; \
     do echo -n "$f: "; \
     rwfileinfo --fields=count-records $f; \
     done
incoming-server.raw: incoming-server.raw:
  count-records       1001
incoming-client.raw: incoming-client.raw:
  count-records       41
outgoing-server.raw: outgoing-server.raw:
  count-records       41
outgoing-client.raw: outgoing-client.raw:
  count-records       1002
leftover.raw: leftover.raw:
  count-records       2
```

Example 4.3: Complex Manifold to Select Inbound Client and Server Flows

- The first `rwfilter` command in the manifold filters out unwanted traffic.

  `--start` and `--sensor` selection specify the time period and network sensor of interest.

  `--protocol=6` selects Transmission Control Protocol (TCP) traffic.

  `--packets=3-` selects flow records with three or more packets indicating TCP sockets.

`--pass=stdout` sends all records that match the filtering criteria to the second part of the manifold. Records that do not match are ignored.

- The second `rwfilter` command in the manifold filters out inbound server traffic.

  `stdin` tells the `rwfilter` command to process flow records from standard input (i.e., the output from the previous `rwfilter` command).

  `--type=in` selects inbound traffic.

  `--flags-initial=SA/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN/ACK", indicating that the source IP address is a server.

  `--pass=incoming-server.raw` stores flow records that match the filtering criteria to the file `incoming-server.raw`

  `--fail=stdout` sends all records that do not match the filtering criteria to the next part of the manifold.

- The third `rwfilter` command in the manifold filters out inbound client traffic.

  `--type=in` selects inbound traffic.

  `--flags-initial=S/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN", indicating the source IP address is a client.

  `--pass=outgoing-client.raw` stores flow records that match the filtering criteria to the file `outgoing-client.raw`.

- The fourth `rwfilter` command in the manifold filters out outbound server traffic.

  `--type=out` selects outbound traffic.

  `--flags-initial=SA/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN/ACK", indicating the source IP address is a server.

  `--pass=outgoing-server.raw` stores flow records that match the filtering criteria to the file `outgoing-server.raw`

- The fifth (and final) `rwfilter` command in the manifold filters out outbound client traffic.

  `--type=out` selects outbound traffic.

  `--flags-initial=S/SA` selects traffic where TCP flags on the first packet of the flow record are set to "SYN", indicating the source IP address is a client.

  `--pass=outgoing-client.raw` stores flow records that match the filtering criteria to the file `outgoing-client.raw`

  `--fail=leftover.rw` saves all records that do not match the filtering criteria in the file `leftover.raw`.

4.2.2 Finding Low-Packet Flows with `rwfilter`

Raj is an analyst who's investigating illegitimate network traffic. He's decided to focus on short TCP flows, particularly those that don't involve just SYN flags (which might be attempts to connect with no response) or just RST flags (which might be redundant termination).

The TCP state machine is complex. Legitimate service requests require a minimum of three (more commonly four) packets in the flow from client to server. Flows from server to client may only have two packets.

Raj knows that several types of illegitimate traffic (such as port scans and responses to spoofed-address packets) involve TCP flow records with low numbers of packets. Although legitimate TCP flow records occasionally have low numbers of packets (such as continuations of previously timed-out flow records, contact attempts on hosts that do not exist or are down, services that are not configured, and RST packets on already closed connections), this behavior is relatively rare. Understanding where low-packet TCP network traffic comes from and when such flow records are collected most frequently can help him to identify traffic that is potentially illegitimate.

Example 4.4 shows how Raj applies the manifold concept from Section 4.2.1 to find low-packet traffic.

1. Raj's first call to `rwfilter` in Command 1 selects all incoming flow records in the repository on 6/17/2015, that describe TCP traffic, and that had one to three packets in the flow record. It is the only `rwfilter` call that pulls data directly from the SiLK repository; as such, it is the only one that uses selection parameters.

   This call also uses a combination of partitioning parameters (`--protocol` and `--packets`) to isolate low-packet TCP flow records from the selected time range. It uses the `--pass` switch twice: once to save the selected records to the file `/.Ex4-data/lowpacket.rw` and once to direct the selected records to standard output (`stdout`) for use by the next `rwfilter` command in the manifold.

   The `--print-statistics` parameter saves information about the `rwfilter` call to the file `temp-all.txt`, including the number of repository files retrieved by `rwfilter`, the total number of flow records, and the number of records that passed or failed the filter.

2. The second call to `rwfilter` in Command 1 uses `--flags-all` as a partitioning parameter to pull out flow records of interest. It passes flow records that had the SYN flag set in any of their packets, but do not have the ACK, RST, and FIN flags set in any of their packets. It fails those that did not show this flag combination. It saves statistics to the file `temp-syn.txt`. Records that fail this filter are passed to the next `rwfilter` call via the `--fail=stdout` parameter.

3. The third call to `rwfilter` extracts the flow records that have the RST flag set, but had the SYN and FIN flags not set. It saves statistics to the file `temp-rst.txt`.

4. Command 2 displays the statistical information saved at each step in the manifold.

By using a manifold, Raj progressively refines his initial data pull to isolate low-packet flows of interest. He can now investigate these flows for illegitimate activity.

4.2.3 Time Binning, Options, and Thresholds With `rwstats`, `rwuniq` and `rwcount`

Approximating Flow Behavior Over Time

SiLK flow records do not contain information about the time distribution of packets and bytes during a flow. Grouping packets into flow records results in a loss of timing information; specifically, it is not possible to

```
<1>$ rwfilter --start-date=2015/06/17 \
   --type=in,inweb --protocol=6 --packets=1-3 \
   --print-statistics=temp-all.txt \
   --pass=./Ex4-data/lowpacket.rw --pass=stdout \
| rwfilter --flags-all=S/SARF \
   --print-statistics=temp-syn.txt \
   --pass=./Ex4-data/synonly.rw --fail=stdout stdin \
| rwfilter --flags-all=R/SRF \
   --print-statistics=temp-rst.txt \
   --pass=./Ex4-data/reset.rw stdin
<2>$ cat temp-all.txt output/temp-syn.txt output/temp-rst.txt
Files   415.  Read    9083613.  Pass     906016. Fail     8177597.
Files     1.  Read     906016.  Pass      97279. Fail      808737.
Files     1.  Read     808737.  Pass     130145. Fail      678592.
```

Example 4.4: Extracting Low-Packet Flow Records

tell how the packets in a flow are distributed over time. Even at the sensor, the information about the time distribution of packets in a flow is lost.

By default, SiLK distributes the packets and bytes equally across all the milliseconds in the flow's duration. This approximation works well for investigating overall trends in network behavior.

However, some types of analysis benefit from intentionally changing the time distribution of packets. For example, incident analysis investigates behavior during specific time bands. Changing the time distribution of packets and bytes (or *load-scheme*) emphasizes different flow characteristics of interest.

Analysts can impose a specific time distribution on `rwcount` by using the `--load-scheme` parameter. `rwcount` can assign one of seven time distributions of packets and bytes in the flow to allocate the volume to time bins.

Figure 4.5 illustrates the allocation of flows, packets, and bytes to time bins under different load-schemes (which are described more fully in Table 4.1). The squares depict a fixed number of packets or bytes. The partial squares are proportional to the complete squares. The wide, solidly filled rectangles depict entire flows, along with their packets and bytes. They appear once in schemes 1, 2, and 3 (where one bin receives the entire flow with its packets and bytes) and in every bin for scheme 5 (where every bin receives the entire flow). The wide, hollow rectangles only appear in scheme 6 and represent whole flows with no packets or bytes.

Using Thresholds to Profile a Slice of Flows

`rwuniq` allows you to set thresholds to segregate IP addresses by the number of flows, sizes of flows, and other values. Recall that `rwuniq` reads SiLK flow records, groups them according to a key composed of user-specified flow attributes, then computes summary values for each group (or bin), such as the sum of the bytes fields for all records that match the key. Thresholding limits the output of `rwuniq` to bins where the summary value meets user-specified minimum or maximum values.

The `--bytes`, `--packets`, and `--flows` parameters are all threshold operators for filtering. For example, to show only the source IP addresses with 200 flow records or more, use the `--flows=200-` parameter as shown in Example 4.5.

Value	Parameter Name	Volume Allocation	Guidelines for Use
0	`bin-uniform`	Allocates equal parts of volume to every bin in timespan	Computes the average load per bin, smoothing out peaks and valleys.
1	`start-spike`	Stores entire volume in the first millisecond of flow (i.e., the flow's `stime` bin)	Emphasizes the onset of periodic behavior. Puts all packets and bytes into one bin even if the flow spans multiple bins.
2	`end-spike`	Stores entire volume in the last millisecond of the flow (i.e., the flow's `etime` bin)	Emphasizes flow termination. Puts all packets and bytes into one bin even if the flow spans multiple bins.
3	`middle-spike`	Stores entire volume in the middle millisecond of the flow	Emphasizes payload transfer. Puts all packets and bytes into one bin even if the flow spans multiple bins.
4	`time-proportional`	Proportionally allocates the flow's bytes, packets, and record count (1 for one flow) to all bins in the flow according to how much time the flow spent in each bin's timespan	**Default load scheme; recommended for most analyses.** Gives the average load per time period. Smooths out peaks and valleys over time.
5	`max-volume`	Assigns entire flow volume to each bin	Overestimates load; computes worst-case scenario for service loading.
6	`min-volume`	Assigns one flow to each bin	Underestimates load; computes best case scenario for service loading.

- For `--load-scheme` values `0` through `4`, the flow record count adds up to 1. The byte and packet counts add up to the counts in the flow record.
- For `--load-scheme` values `5` and `6`, the flow record count does not add up to 1. The byte and packet counts do not add up to the counts in the flow record.

Table 4.1: Time distribution options for `rwcount --load-scheme`

```
<1>$ rwfilter --start-date=2015/06/17T15 --type=in --protocol=0- \
   --pass=./Ex4-data/in_month.rw
<2>$ ls -l ./Ex4-data/in_month.rw
-rw-r--r--. 1 analyst analyst 7409538 Apr 25 17:22 ./Ex4-data/in_month.rw
<3>$ rwuniq ./Ex4-data/in_month.rw --field=sIP --value=Flows \
   --flows=200- \
| head -n 10
            sIP|   Records|
 192.168.143.57|       481|
 192.168.161.26|       443|
  192.168.40.51|       254|
     10.0.40.23|      5128|
 192.168.165.83|       378|
  192.168.40.25|     11951|
192.168.162.160|       401|
     10.0.40.92|      1838|
192.168.143.162|       635|
```

Example 4.5: Constraining Counts to a Threshold by using `rwuniq --flows`

In addition, `rwuniq` can count bytes and packets for a flow threshold through the `bytes` and `packets` values in the `--values` parameter, as shown in Example 4.6. This example counts the byte and packet volumes for all IP addresses that exceed a minimum flow threshold of 2000 records (`--values=Bytes,Packets,Flows --flows=2000-`).

```
<1>$ rwfilter --start-date=2015/06/17T15 --type=in --protocol=0- \
   --pass=./Ex4-data/in_month.rw
<2>$ ls -l ./Ex4-data/in_month.rw
-rw-r--r--. 1 analyst analyst 7409538 Apr 25 17:22 ./Ex4-data/in_month.rw
<3>$ rwuniq ./Ex4-data/in_month.rw --field=sIP \
   --values=Bytes,Packets,Flows --flows=2000- \
| head -n 10
            sIP|               Bytes|        Packets|   Records|
     10.0.40.23|             9331109|         108622|      5128|
  192.168.40.25|             6066944|          58819|     11951|
  192.168.20.58|             4128957|          54314|     28189|
     10.0.40.53|            64029248|         281180|     33917|
 192.168.200.10|             6089612|          20535|      8816|
     10.0.40.54|             6131060|          42785|      7075|
     10.0.40.20|            36120528|         345466|    149284|
     10.0.20.58|             4043680|          55644|     21253|
     67.215.0.8|          1499433280|        6096359|     30549|
```

Example 4.6: Setting Minimum Flow Thresholds with `rwuniq --values`

If a range (such as `--flows=2000-`) is not specified, the parameter simply adds the named count to the list started by the `--values` parameter. We recommend using the `--values` parameter for this purpose. `--values` provides greater control over the order in which the values are displayed than the other thresholding parameters.

Hint 4.3: How to Specify Ranges with `rwuniq`

`rwuniq` provides three ways to specify ranges: with the low and high bounds separated by a hyphen (e.g., 200–2000); with a low bound followed by a hyphen (e.g., 200–) denoting that there is no upper bound; and with a low bound alone (e.g., 200). Unlike `rwfilter` partitioning values, the last method denotes a range with no upper bound, not just a single value. **We do not recommend using this method because it can lead to confusion.**

If multiple threshold parameters are specified, `rwuniq` will print all records that meet *all* of the threshold criteria, as shown in Example 4.7.

```
<1>$ rwfilter --start-date=2015/06/17T15 --type=in --protocol=0- \
   --pass=./Ex4-data/in_month.rw
<2>$ ls -l ./Ex4-data/in_month.rw
-rw-r--r--. 1 analyst analyst 7409538 Apr 25 17:23 ./Ex4-data/in_month.rw
<3>$ rwuniq ./Ex4-data/in_month.rw --field=sIP \
   --values=Bytes,Packets,Flows --flows=2000- \
   --packets=100000-
            sIP|               Bytes|         Packets|   Records|
     10.0.40.23|             9331109|          108622|      5128|
     10.0.40.53|            64029248|          281180|     33917|
     10.0.40.20|            36120528|          345466|    149284|
     67.215.0.8|          1499433280|         6096359|     30549|
  192.168.40.20|            67729921|          283002|    106848|
```

Example 4.7: Constraining Flow and Packet Counts with `rwuniq --flows` and `--packets`

Profiling With Compound Keys

Profiling can also be done by counting and thresholding combinations of fields, in addition to the simple counting shown previously. Both the `rwuniq` and `rwstats` commands support compound keys.

Using Compound Keys with `rwuniq`. To use a compound key, specify it using a comma-separated list of values or ranges in the `rwuniq --fields` parameter. Keys can be manipulated in the same way as with `rwcut`: `--fields=3,1` is a different key than `--fields=1,3`.

In Example 4.8, the `--fields` parameter is used to identify communication between clients and specific services only when the number of flows for the key exceeds a threshold. It counts and thresholds incoming traffic to identify those source IP addresses with the highest number of flow records that connect to specific TCP ports (`--fields=sIP,sPort`).

Using Compound Keys with `rwstats`. Alternatively, you can perform this analysis by running the `rwstats` command with compound keys. In Example 4.9, the `--fields` parameter is similarly used to count incoming traffic and identify the eleven source IP addresses with the highest number of flow records that connect to specific TCP ports (`--fields=sIP,sPort`). In addition to counting and displaying these records, `rwstats` computes their cumulative statistics. This allows you to directly compare the amount of traffic carried by each source IP-port combination.

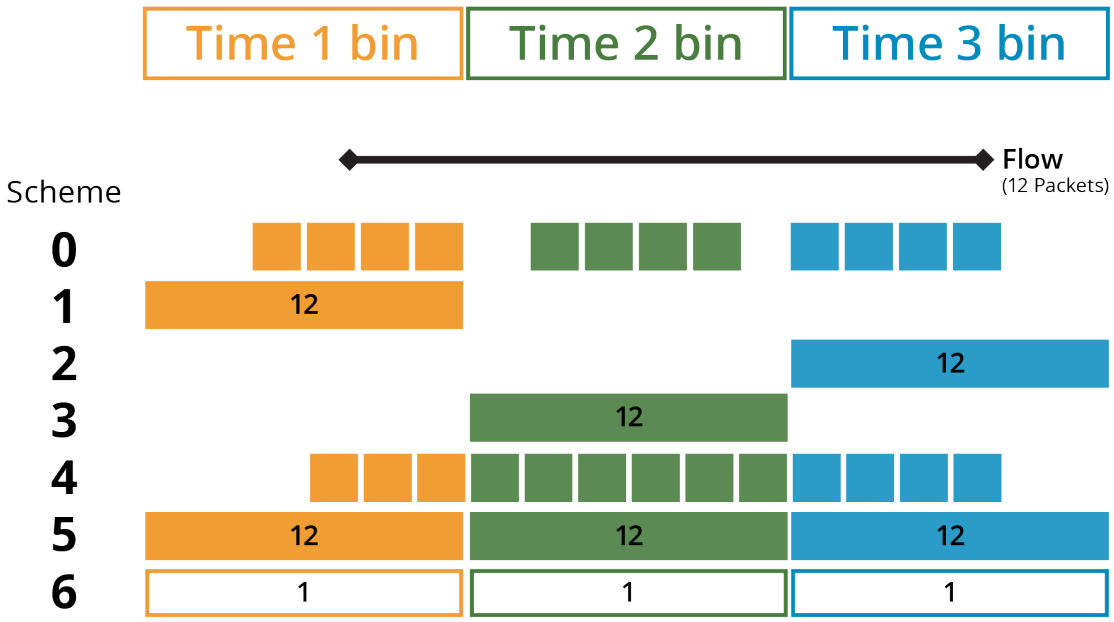

Figure 4.5: Allocating Flows, Packets and Bytes via `rwcount` Load-Schemes

```
<1>$ rwfilter --start-date=2015/06/17T15 --type=in --protocol=6 \
   --pass=./Ex4-data/in_month.rw
<2>$ ls -l ./Ex4-data/in_month.rw
-rw-r--r--. 1 gtsanders domain users 2059092 May 14 12:15 ./Ex4-data/in_month.rw
<3>$ rwuniq --fields=sIP,sPort --value=Flows --flows=20- \
   ./Ex4-data/in_month.rw \
| head -n 11
            sIP|sPort|   Records|
192.168.143.162|  591|        26|
192.168.111.131|  591|        28|
192.168.122.141|  591|        26|
     10.0.40.20|   88|       476|
     10.0.40.20|  139|      1189|
192.168.164.119|  591|        26|
  192.168.40.20|60309|        20|
  192.168.40.25|  445|       120|
192.168.165.216|  591|        26|
  192.168.40.20|  135|       265|
```

Example 4.8: Profiling IP addresses with `rwuniq --fields`

```
<1>$ rwfilter --start-date=2015/06/17T15 --type=in --protocol=6 \
   --pass=./Ex4-data/in_month.rw
<2>$ ls -l ./Ex4-data/in_month.rw
-rw-r--r--. 1 analyst analyst 2059092 May 14 12:26 ./Ex4-data/in_month.rw
<3>$ rwstats --count=11 --fields=sIP,sPort --value=Flows \
   ./Ex4-data/in_month.rw \
INPUT: 124356 Records for 11637 Bins and 124356 Total Records
OUTPUT: Top 11 Bins by Records
            sIP|sPort|   Records|  %Records|   cumul_%|
     10.0.40.53| 5723|     33103| 26.619544| 26.619544|
     67.215.0.8|11009|     12479| 10.034900| 36.654444|
     67.215.0.8|11007|     10714|  8.615588| 45.270031|
     67.215.0.8|  135|      7181|  5.774550| 51.044582|
     10.0.40.23| 8443|      5128|  4.123645| 55.168227|
  192.168.40.20|   88|      3296|  2.650455| 57.818682|
     10.0.40.20|  445|      1986|  1.597028| 59.415710|
     10.0.40.20|  389|      1367|  1.099263| 60.514973|
     10.0.40.20|  139|      1189|  0.956126| 61.471099|
  192.168.70.10| 8082|      1077|  0.866062| 62.337161|
     10.0.40.20|49158|      1071|  0.861237| 63.198398|
```

Example 4.9: Profiling IP addresses with `rwstats --fields`

Isolating Behaviors of Interest

`rwuniq` can be used in conjunction with `rwfilter` to profile flow records for a variety of behaviors:

1. Use `rwfilter` to filter records for the behavior of interest. (To filter for multiple behaviors, set up a manifold as described in Section 4.2.1.)
2. Use `rwuniq` to count the records that exhibit that behavior.

This can help you to understand the behavior of hosts that use or provide a variety of services. Example 4.10 shows how to generate data that compare hosts showing DNS (domain name system) and non-DNS behavior among a group of flow records. We can find DNS servers by filtering the file `in_month.rw` (created in Example 4.8) for hosts that carry traffic on port 53 with the UDP protocol (17).

1. Command 1 first isolates the set of hosts of interest by using `rwfilter` to filter records from `in_month.rw` with UDP traffic on port 53 (`--protocol=17 --aport=53`). It then uses `rwset` to generate an IPset (`interest.set`) from the records that pass the filter.
2. Command 2 uses `interest.set` to filter `in_month.rw` again to distinguish IP addresses with general UDP traffic from the set of hosts that carry DNS traffic on port 53. It then uses `rwuniq` to count the DNS flow records and sorts them by source address.

   Although `rwuniq` will correctly sort the output rows by IP address without zero-padding, the upcoming `join` command will not understand that the input is properly sorted without `rwuniq` first preparing the addresses with the `--ip-format=zero-padded` parameter.
3. Command 3 counts the non-DNS flow records created with the `--fail` switch in Command 2 and sorts them by source address.

4. Command 4 merges the two count files by source address and then sorts them by number of DNS flows with the results shown. Hosts with high counts in both columns should be either workstations or gateways. Hosts with high counts in DNS and low counts in non-DNS should be DNS servers.[10]

For more complex summaries of behavior, use the `rwbag` command and its related utilities as described in Section 4.2.4.

```
<1>$ rwfilter in_month.rw --protocol=17 --aport=53 --pass=stdout \
| rwset --sip-file=interest.set
<2>$ rwfilter in_month.rw --sipset=interest.set --protocol=17 \
   --pass=stdout \
| rwfilter stdin --aport=53 --fail=not-dns.rw --pass=stdout \
| rwuniq --fields=sIP --no-titles --ip-format=zero-padded \
   --sort-output --output-path=dns-saddr.txt
<3>$ rwuniq not-dns.rw --fields=sIP --no-titles \
   --ip-format=zero-padded --sort-output \
   --output-path=not-dns-saddr.txt
<4>$ echo '             sIP|       DNS||   not DNS|' \
; join -t'|' dns-saddr.txt not-dns-saddr.txt \
| sort -t'|' -nrk2,2 \
| head -n 5
             sIP|       DNS||   not DNS|
010.000.040.020|    124652||     14322|
192.168.040.020|     98128||       797|
192.168.200.010|      7123||        64|
192.168.040.025|      5188||      5127|
192.168.165.216|       508||        47|
```

Example 4.10: Isolating DNS and Non-DNS Behavior with `rwuniq`

4.2.4 Summarizing Network Traffic with Bags

IPsets contain lists of IP addresses. However, it's often useful to associate a value with each address in an IPset. For instance, you may want to associate the IP addresses that engage in web traffic with the volume of flows, packets, or bytes of web traffic that each address carries. *Bags* are extended sets that contain these types of key-value pairs.

Where IPsets record the presence or absence of key values, bags add the ability to count the number of instances of a particular key value—that is, the number of bytes, the number of packets, or the number of flow records associated with that key. Bags also allow the analyst to summarize traffic on characteristics other than IP addresses—specifically on protocols and ports.[11]

Bags can be thought of as enhanced IPsets. Like IPsets, they are binary structures that can be manipulated using a collection of tools. As a result, operations that are performed on IPsets have analogous bag operations, such as addition (the equivalent to union). Analysts can also extract a cover set (the set of all IP addresses in the bag) for use with `rwfilter` and the IPset tools.

[10] The full analysis to identify DNS servers is more complex and will not be dealt with in this handbook.

[11] PySiLK allows for even more general bag key values and count values. See the documentation *PySiLK: SiLK in Python* for more information.

Generating Bags from Network Flow Data

The `rwbag` command creates bags from raw network flow data, either directly output from the `rwfilter` command or stored in a file. The *key* parameter specifies the network flow record field that serves as the key for the bag. Examples of keys include source IP address (`sIPv4`, `sIPv6`), destination IP address (`dIPv4`, `dIPv6`), source port (`sPort`), destination port (`dport`), `protocol`, `packets`, and `bytes`.

The *counter* parameter sums up the number of `records`, `flows`, `packets`, or `bytes` for the flow record field specified by *key*. *outputfile* is the name of the file where the bag is stored, such as `mybag.bag`. Bags are stored in binary format to make analysis tasks faster and more efficient. Use the `rwbagcat` command to display the contents of a bag.

Example 4.11 shows an example of how to use `rwbag` in conjunction with `rwfilter`. You may specify multiple `--bag-file` parameters when you issue the `rwbag` command.

```
<1>$ rwfilter --type=in,inweb --start-date=2015/06/18T12 \
   --protocol=6 --pass=stdout \
| rwbag --sip-packets=x.bag --dip-flows=y.bag
<2>$ file x.bag y.bag
x.bag: data
y.bag: data
```

Example 4.11: Generating Bags with `rwbag`

Help with `rwbag`. For a list of command options, type `rwbag --help` at the command line. For more information about the `rwbag` command, type `man rwbag`.

Using Bags to Summarize Web Traffic.

To show how useful bags can be, we return to the task mentioned earlier in this section: analyzing outgoing web traffic. We want to find out which IP addresses engage in web traffic and will narrow our study to outgoing TCP flows on ports 80 and 443. We can find these IP addresses by using the `rwfilter` and `rwuniq` commands as shown in Example 4.12.

```
<1>$ rwfilter --start-date=2015/06/17 --sensors=S1 --type=outweb \
   --protocol=6 --sport=80,443 --packets=3- --pass=stdout \
| rwuniq --fields=sip --values=bytes
            sIP|               Bytes|
  192.168.40.24|              877782|
  192.168.20.59|             1392516|
  192.168.40.91|              124548|
  192.168.40.92|              124548|
```

Example 4.12: Summarizing Network Traffic with `rwuniq`

This provides us with addresses and byte counts in text format. However, we would like to use this information during further analysis with SiLK—for example, as an IPset with the addresses and some way to store the number of bytes for each address. To store such a list, we need to create a bag with the `rwbag` command as shown in Example 4.13. We want the key to be the IP address and the value to be the number of bytes of outbound web traffic.

```
<1>$ rwfilter --start-date=2015/06/17 --sensors=S1 --type=outweb \
   --protocol=6 --sport=80,443 --packets=3- --pass=stdout \
| rwbag --bag-file=sipv4,sum-bytes,outgoingweb.bag \
<2>$ file outgoingweb.bag
outgoingweb.bag: SiLK, RWBAG v3, Little Endian, LZO compression
<3>$ rwbagcat outgoingweb.bag
  192.168.20.59|             1392516|
  192.168.40.24|              877782|
  192.168.40.91|              124548|
  192.168.40.92|              124548|
```

Example 4.13: Summarizing Network Traffic with Bags

The file `outgoingweb.bag` contains the list of IP addresses that carry outgoing web traffic and the volume of outbound traffic in bytes that flows through each address. Unlike the output of the `rwuniq` command, this information is stored in a single binary file, as shown in 4.13. We can now use this file during further analysis of outbound web traffic.

Generating Bags From IP Sets or Text

You can create a bag from an existing set or a text file by using the `rwbagbuild` tool. This allows you to associate counts with items in a set or file. It also gives you more flexibility with creating bags than the `rwbag` command does. For instance, you can use `rwbagbuild` to count something other than bytes, packets, or flow records for an address.

`rwbagbuild` takes either an IPset (as specified in `--set-input`) or a text file (as specified in `--bag-input`), but not both.

- For IPset input, the `--default-count` parameter specifies the count value for each set element in the output bag. If no `--default-count` value is provided, the count will be set to one.
- For text-file input, the lines of the file are expected to consist of a key value, a delimiter (by default the vertical bar), and a count value. Keys can be IP addresses (including canonical forms, CIDR blocks, and SiLK address wildcards) or unsigned integers.

Help with `rwbagbuild`. For a list of command options, type `rwbagbuild --help` at the command line. For more information about the `rwbagbuild` command, type `man rwbagbuild`.

Detecting Scanning Activity with `rwbagbuild` and `rwscan`

Example 4.14 shows how to use the `rwbagbuild` command in conjunction with the `rwscan` command to create a bag that contains IP addresses that show evidence of scanning activity and the number of flows associated with them.

`rwscan` analyzes SiLK flow records for signs of network scanning—when an external host gathers information about a network during the reconnaissance phase of an attack. It takes sorted network flow records as input and outputs in columnar text format any IP addresses that show signs of network scanning. This is useful for identifying hosts that are conducting reconnaissance and the ports and protocols of interest to them. Pairing this command with `rwbagbuild` allows you to create a bag that stores scanner IP addresses and attributes of their activity for further investigation.

1. Command 1 uses `rwfilter` to pull inbound TCP traffic (`--proto=6 --type=in,inweb`).
2. The `rwscan` command requires input to be pre-sorted by source IP address, protocol, and destination IP address. Command 1 therefore calls `rwsort --fields=sip,proto,dip` to sort the selected records.
3. Command 1 then uses `rwscan` to search for IP addresses that show signs of scanning activity. It pipes the output through the operating system `cut` command to remove the delimiters (`|`) in the `rwscan` output.
4. Finally, Command 1 uses the `rwbagbuild` command to create a bag (`scanners.bag`) from the `rwscan` output. It uses the scanning IP addresses as the key and the number of flow records as the associated count for the bag entries.
5. Commands 2 and 3 display the list of scanners created in Command 1. Command 2 uses the `rwbagcat` command to create a text file that contains the contents of `scanners.bag`. (`rwbagcat` is described later in this section.) Command 3 shows that file.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --proto=6 --type=in,inweb --pass=stdout \
| rwsort --fields=sip,proto,dip \
| rwscan --scan-model=2 --output-path=stdout --no-title \
| cut -f1,5 -d'|' \
| rwbagbuild --bag-input=stdin --key-type=sIPv4 \
   --counter-type=records >scanners.bag
<2>$ echo 'Scanner        |               Flows|' >scanners.txt \
; rwbagcat scanners.bag >>scanners.txt
<3>$ cat scanners.txt
Scanner        |               Flows|
  192.168.181.8|               45428|
```

Example 4.14: Creating a Bag of Network Scanners with `rwbagbuild` and `rwscan`

Help with `rwscan`. For a list of command options, type `rwscan --help` at the command line. For more information about the `rwscan` command, type `man rwscan`.

Specifying IP addresses with `rwbagbuild`. The `rwbagbuild` command does not support mixed input of IP addresses and integer values, since there is no way to specify whether the number represents an IPv4 address or an IPv6 address. (For example, does `1` represent `::FFFF.0.0.0.1` or `::1`?) `rwbagbuild` also does not support symbol values in its input, so some types commonly expressed as symbols (TCP flags, attributes) must be translated into an integer form.

Similarly, `rwbagbuild` does not support formatted time strings. Times must be expressed as unsigned integer seconds since UNIX epoch (i.e., the number of seconds since midnight, January 1, 1970). If the delimiter character is present in the input data, it must be followed by a count. If the `--default-count` parameter is used, its argument will override any counts in the text-file input; otherwise the value in the file will be used. If no delimiter is present, either the `--default-count` value will be used or the count will be set to 1 if no such parameter is present. If the key value cannot be parsed or a line contains a delimiter but no count, `rwbagbuild` prints an error and exits.

Displaying the Contents of Bags

To view or summarize the contents of a bag, use the `rwbagcat` command. By default, it displays the contents of a bag in sorted order as shown in Example 4.15.

```
<1>$ rwbagcat x.bag \
      | head -n 5
    192.0.2.198|                1281|
    192.0.2.227|                  12|
    192.0.2.249|                  90|
 198.51.100.227|                   3|
 198.51.100.244|                 101|
```

Example 4.15: Viewing the Contents of a Bag with `rwbagcat`

Help with `rwbagcat`. For a complete list of command options, type `rwbagcat --help` at the command line. For more information about the `rwbagcat` command, type `man rwbagcat`.

Thresholding Bags. In Example 4.15, the *counts* (the number of elements that match a particular IP address) are printed per key. `rwbagcat` can also print values within ranges of both counts and keys, as shown in Example 4.16.

```
<1>$ rwbagcat --mincounter=100 --maxcounter=600 kerbserv.bag
     10.0.40.20|                 574|
     67.215.0.5|                 245|
<2>$ rwbagcat --minkey=10.0.0.0 --maxkey=192.168.255.255 \
   kerbserv.bag
     10.0.40.20|                 574|
     67.215.0.5|                 245|
  192.168.40.20|                4596|
```

Example 4.16: Thresholding Results with `rwbagcat --mincounter`, `--maxcounter`, `--minkey`, and `--maxkey`

These thresholding values can be used in any combination.

Counting Keys in Bags. In addition to thresholding, `rwbagcat` can also reverse the index; that is, instead of printing the number of counted elements per key, it can produce a count of the number of keys matching each count using the `--bin-ips` parameter. This reverse count is shown in Example 4.17.

- In Command 1, it is shown using linear binning—one bin per value, with the counts showing how many keys had that value.
- In Command 2, it is shown with binary binning—values collected by powers of two and with counts of keys having summary volume values in those ranges.
- In Command 3, it is shown with decimal logarithmic binning—values collected in bins that provide one bin per value below 100 and an even number of bins for each power of 10 above 100, arranged logarithmically and displayed by midpoint. This option supports logarithmic graphing options.

```
<1>$ rwbagcat --bin-ips dns.bag \
| head -n 5
                   1|                    1|
                   3|                    1|
                  14|                    1|
                  18|                    1|
                  30|                    1|
<2>$ rwbagcat --bin-ips=binary dns.bag \
| head -n 5
      2^00 to 2^01-1|                    1|
      2^01 to 2^02-1|                    1|
      2^03 to 2^04-1|                    1|
      2^04 to 2^05-1|                    2|
      2^09 to 2^10-1|                    3|
<3>$ rwbagcat --bin-ips=decimal dns.bag \
| head -n 5
                   1|                    1|
                   3|                    1|
                  14|                    1|
                  18|                    1|
                  30|                    1|
```

Example 4.17: Displaying Unique IP Addresses per Value with `rwbagcat --bin-ips`

The `--bin-ips` parameter can be particularly useful for distinguishing between sites that are hit by scans (where only one or two packets may appear) from sites that are engaged in legitimate activity.

Formatting Key Values for Bags. If the bag is not keyed by IP address, the optional `--key-format` parameter makes it *much* easier to read the output of `rwbagcat`. Example 4.18 shows the difference in output for a sIP-keyed bag counting bytes, where the IP addresses are shown in decimal and hexadecimal formats. `--key-format` is only necessary if the key has a "custom" format.

Comparing the Contents of Bags

Once you have created bags to store key-value pairs, you can compare their contents to identify common values and trends. For example, you may want to compare the traffic volumes associated with two groups of IP addresses, each in a separate bag file. Use the `rwbagtool --compare` parameter to compare the contents of two bags. It stores the output from this comparison in a new bag file.

```
<1>$ rwbagcat kerbserv.bag
     10.0.40.20|              751382|
     67.215.0.5|              395218|
  192.168.40.20|             5510424|
<2>$ rwbagcat --key-format=decimal kerbserv.bag
 167782420|              751382|
1138163717|              395218|
3232245780|             5510424|
<3>$ rwbagcat --key-format=hexadecimal kerbserv.bag
 a002814|              751382|
43d70005|              395218|
c0a82814|             5510424|
```

Example 4.18: Displaying Decimal and Hexadecimal Output with `rwbagcat --key-format`

For each *key* that appears in both bag files, the `--compare` option compares the value of the key's associated *counter* (i.e., the number of bytes, packets, or records summed up by the `rwbag` or `rwbagbuild` command) in the first file to the value of the key's counter in the second file.

- If the comparison is *true*, the key appears in the resulting bag file with a counter of 1.
- If the comparison is *false*, the key is not present in the output file.
- Keys that appear in only one of the input bag files are ignored.

`rwbagtool --compare` can perform the following comparisons:

`lt` Finds keys in the first bag file whose counters are less than those in the second bag file.

`le` Finds keys in the first bag file whose counters are less than or equal to those in the second bag file.

`eq` Finds keys whose counters are equal in both files.

`ge` Finds keys in the first bag file whose counters are greater than or equal to those in the second bag file.

`gt` Finds keys in the first bag file whose counters are greater than those in the second bag file.

Help with `rwbagtool`. For a list of command options, type `rwbagtool --help` at the command line. For more information about the `rwbagtool` command, type `man rwbagtool`.

4.2.5 Working with Bags and IPsets

Since bags are essentially enhanced IPsets, SiLK provides operations that enable you to create IPsets from bags and compare the contents of bags with those of IPsets.

Extracting IPsets from Bags

Sometimes you will want to extract an IPset from a bag—for instance, if you used the `rwbagtool --compare` command to compare traffic associated with IP addresses in two bags and want to analyze the set of IP addresses in the new bag. Use the `rwbagtool --coverset` parameter to generate a *cover set*: the set of IP addresses in a bag. The resulting IPset file can be used with `rwfilter` and manipulated with any of the `rwset` commands.

Example 4.19 shows how to extract an IPset from a bag file. The `rwsetcat` command displays the contents of the resulting set and the `rwbagcat` command displays the contents of the original bag—showing that the IP addresses that comprise the set are identical to those in the bag.

```
<1>$ rwbagtool outgoingweb.bag --coverset \
   --output-path=outgoingweb.set
<2>$ rwsetcat outgoingweb.set \
| head -n 3
192.168.20.59
192.168.40.24
192.168.40.91
<3>$ rwbagcat outgoingweb.bag \
| head -n 3
  192.168.20.59|              1392516|
  192.168.40.24|               877782|
  192.168.40.91|               124548|
```

Example 4.19: Creating an IP Set from a Bag with `rwbagtool --coverset`

Hint 4.4: How to Use `rwbagtool --coverset` with Bag Files

Be careful of bag contents when using `rwbagtool --coverset`. Since `rwbagtool` does not limit operations by the type of keys contained within a bag, the `--coverset` parameter will interpret the keys as IP addresses even if they are actually protocol or port keys. This will lead to confusion and analysis errors!

Intersecting Bags and IPsets

You may want to find out whether the IP addresses in an IPset are also contained in a bag. You may also want to limit the contents of a bag to a specific group of IP addresses—for instance, those on a specific subnet.

The `rwbagtool --intersect` and `--complement-intersect` parameters are used to intersect an IPset with a bag. Example 4.20 shows how to use these parameters to extract a specific subnet.

4.2.6 Masking IP Addresses

When working with IP addresses and utilities such as `rwuniq` and `rwstats`, you may want to analyze activity across *networks* rather than individual IP addresses. For example, you may wish to examine all of the activity originating from the /24s constituting the enterprise network rather than generating an individual entry for

```
<1>$ echo '10.0.20.x' >f.set.txt
<2>$ rwsetbuild f.set.txt f.set
<3>$ rwbagtool x.bag --intersect=f.set --output-path=xf.bag
<4>$ rwbagcat x.bag
     10.0.20.58|                 522|
     10.0.20.59|                1652|
    67.215.0.55|                  88|
   117.34.28.84|                  12|
     155.6.3.10|                  30|
     155.6.4.10|                  30|
 192.168.200.10|                3913|
<5>$ rwbagcat xf.bag
     10.0.20.58|                 522|
     10.0.20.59|                1652|
```

Example 4.20: Using `rwbagtool --intersect` to Extract a Subnet

each address. To perform this type of analysis, use the `rwnetmask` command to reduce IP addresses to prefix values of a parameterized length.

The query in Example 4.21 is followed by an `rwnetmask` call to retain only the first 24 bits (three octets) of source IPv4 addresses, as shown by the `rwcut` output.

```
<1>$ rwfilter --type=out,outweb --start-date=2015/06/02 \
   --end-date=2015/06/18 --sensors=S0,S1 --protocol=6 \
   --max-pass-records=3 --pass=stdout \
| rwnetmask --4sip-prefix-length=24 --4dip-prefix-length=24 \
| rwcut --fields=1-5
            sIP|            dIP|sPort|dPort|pro|
      10.0.40.0|  192.168.124.0| 1065|  591|  6|
      10.0.40.0|  192.168.166.0| 1066|  591|  6|
      10.0.40.0|   192.168.40.0|58083|   88|  6|
```

Example 4.21: Abstracting Source IPv4 addresses with `rwnetmask`

As Example 4.21 shows, `rwnetmask` replaces the last 8 bits[12] of the source and destination IP addresses with zero, so all IP addresses in the 10.0.40/24 block (for example) will be masked to produce the same IP address. This replaces both source and destination IP addresses with zero. With `rwnetmask`, an analyst can use any of the standard SiLK utilities on networks in the same way the analyst would use the utilities on individual IP addresses.

Help with `rwnetmask`

For a list of command options, type `rwnetmask --help` at the command line. For more information about the `rwnetmask` command, type `man rwnetmask`.

[12] 32 bits total for an IPv4 address minus the 24 bits specified in the command for the prefix length leaves 8 bits to be masked.

4.2.7 Working With IPsets

Iterative multi-path analyses commonly result in multiple SiLK IPset files. These files are usually named to describe their association with a specific aspect of analysis, such as byte thresholds as discussed in Section 2.2.8. As analyses progress, however, it is necessary to understand how IPsets compare and contrast. `rwsetbuild`, `rwsettool`, and `rwsetmember` are three important SiLK IPset tools that are often used together to identify network infrastructure and traffic flow.

Creating IPsets from Text Files

`rwsetbuild` creates binary IPsets from text input files. It can be used to build reference sets to start an analysis, as previously discussed in Section 2.2.8.

Example 4.22 shows how to build an IPset of the FCCX-15 internal network reference from the `ipblocks` statements in the `sensors.conf` configuration file. Command 1 displays the first five lines of the `monitored_nets.set.txt` textual input file. Command 2 shows the use of the `rwsetbuild` to build the binary IPset from text. Finally, Command 3 shows the file command to verify the `monitored_nets.set` filetype.

```
<1>$ head -n 5 monitored_nets.set.txt
# Text file of monitored networks
# Build from sensor.conf ipblocks
10.0.10.0/24
10.0.20.0/24
10.0.30.0/24
<2>$ rwsetbuild monitored_nets.set.txt monitored_nets.set
<3>$ file monitored_nets.set
monitored_nets.set: SiLK, IPSET v2, Little Endian, LZO compression
```

Example 4.22: Generating a Monitored Address Space IPset with `rwsetbuild`

Example 4.23 shows a similar approach to build an IPset of the broadcast address space with `rwsetbuild`.

```
<1>$ echo 255.255.255.255 >broadcast.set.txt
<2>$ rwsetbuild broadcast.set.txt broadcast.set
<3>$ file broadcast.set
broadcast.set: SiLK, IPSET v2, Little Endian, LZO compression
```

Example 4.23: Generating a Broadcast Address Space IPset with `rwsetbuild`

Help with `rwsetbuild`. For a list of command options, type `rwsetbuild --help` at the command line. For more information about the `rwsetbuild` command, type `man rwsetbuild`.

Manipulating IPsets: DNS Server Example

Once you have constructed SiLK IPsets, use the `rwsettool` command for manipulating them. It provides common algebraic set operations for arbitrary numbers of IPset files.

Example 4.24 shows an example of how to combine two sets to create a comprehensive IPset of the FCCX-15 internal network. It uses the `rwsettool --union` operation to combine both input set files, resulting in a summary set file, `internal_nets.set`. This file represents the FCCX-15 internal network addresses, including IP broadcasts that should not route to public IP space.

```
<1>$ rwsettool --union monitored_nets.set broadcast.set \
   >internal_nets.set
```

Example 4.24: Performing an IPset Union with `rwsettool`

Analysts should determine the time period required for an analysis after creating reference sets. Example 4.25 shows how to use `rwsiteinfo` with the `--fields=repo-start-date,repo-end-date` parameter to determine the full time range of FCCX-15 data: `2015/06/02T13:00:00` to `2015/06/18T18:00:00`.

```
<1>$ rwsiteinfo --fields=repo-start-date,repo-end-date
         Start-Date|           End-Date|
2015/06/02T13:00:00|2015/06/18T18:00:00|
```

Example 4.25: Displaying Repository Dates with `rwsiteinfo`

The dates identified in Example 4.25 are then used as input to the `rwfilter` command to inventory all `out` type Domain Name System (DNS) servers, as shown in Example 4.26. Command 1 shows how to use `rwfilter` to query the entire FCCX-15 repository for DNS servers, which carry `out` type traffic on port 53 (`--dport=53`) with the UDP protocol (`--protocol=17`). It saves those IP addresses to the `dns_servers_out.set` file. Command 2 shows that there were outbound requests to 22 DNS servers on port 53/UDP during the period 2015/06/02 to 2015/06/18.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --protocol=17 --type=out --dport=53 --pass=stdout \
| rwset --dip-file=dns_servers_out.set
<2>$ rwsetcat --count dns_servers_out.set
22
```

Example 4.26: Counting Outbound DNS Servers with `rwset`

Although Example 4.26 generated a comprehensive IPset of outbound DNS servers (`dns_servers_out.set`), it contains all servers that carry `out` type traffic. This means that DNS servers that are contained within the network perimeter may also reside in the resulting IPset. To differentiate between the internal and external network, IP addresses of internal DNS servers must be removed from the set.

Example 4.27 shows how to use the `rwsettool --difference` option to remove the IP addresses of internal DNS servers from the set of DNS servers that reside outside the network perimeter. Command 1 shows how to create the `external_dns_servers.set` set file by finding the difference between the DNS servers contained in `dns_servers_out.set`, but not contained in `internal_nets.set`. Command 2 shows a total of 16 DNS servers that reside external to the network in the FCCX-15 data.

The remaining internal DNS servers can be identified with the `rwsettool --symmetric-difference` option. A symmetric difference is the elements of two sets that are members of either set, but not members of both.

```
<1>$ rwsettool --difference dns_servers_out.set \
   internal_nets.set >external_dns_servers.set
<2>$ rwsetcat --count external_dns_servers.set
16
```

Example 4.27: Finding IPset Differences with `rwsettool`

Example 4.28 shows how the `internal_dns_server.set` set file is generated from a symmetric difference of the `external_dns_servers.set` and `dns_servers_out.set` files, finding a total of 6 internal DNS servers.

```
<1>$ rwsettool --symmetric-difference external_dns_servers.set \
   dns_servers_out.set >internal_dns_servers.set
<2>$ rwsetcat --count internal_dns_servers.set
6
```

Example 4.28: Finding IPset Symmetric Difference with `rwsettool`

Help with `rwsettool`. For a list of command options, type `rwsettool --help` at the command line. For more information about the `rwsettool` command, type `man rwsettool`.

Using Set Membership to Understand Traffic Flow

After you've identified DNS server infrastructure, multi-path analyses may also require identifying traffic flow to specific DNS servers. IPset membership shows how network traffic flows through a network infrastructure.

Finding IPset Membership with `rwsetmember`. Use the `rwsetmember` command to identify if an IP address or pattern is contained within one or more IPset files. `rwsetmember` begins by building per-sensor IPset inventories of outbound traffic to DNS servers, as shown in Example 4.29.

Command 1 shows how to use `rwsiteinfo` to generate a list of sensors for the FCCX-15 dataset. This list of sensors is then used in Commands 2 and 3 of the `dns_servers_by_sensor.sh` script to loop through each sensor name and build an IPset of the DNS servers that are monitored by each sensor.

After creating the per-sensor DNS server IPsets, we can use the `rwsetmember` command as shown in Example 4.30 to identify sensors that monitor specific external DNS servers.

- Command 1 shows how to use `rwsetmember` to identify the sensors that monitor 8.8.x.x DNS servers. The results indicate that sensors `S0`, `S1`, `S2`, `S3`, and `S12` logged DNS requests to 8.8.x.x IP addresses.
- Command 2 shows how these sensors can be combined into a list with `rwsiteinfo` to display their descriptions. The output from the `rwsiteinfo` command shows that DNS clients in the `Div0Ext`, `Div1Ext`, `Div0Int`, `Div1Int1`, and `Div1svc` monitored networks execute DNS queries to 8.8.x.x internet servers.

```
<1>$ rwsiteinfo --fields=sensor:list
                                                                    Sensor:list|
S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21|
<2>$ cat dns_servers_by_sensor.sh
SDATE="2015/06/02"
EDATE="2015/06/18"
SENSORS="S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14"
SENSORS+=" S15 S16 S17 S18 S19 S20 S21"
for SENSOR in $SENSORS; do
  rwfilter --start-date=$SDATE --end-date=$EDATE --type=out \
  --proto=17 --dport=53 --sensor=$SENSOR --pass=stdout \
  | rwset --dip-file="$SENSOR"_dns_servers_out.set
done
<3>$ sh dns_servers_by_sensor.sh
```

Example 4.29: Grouping Outbound DNS Servers by Sensor

```
<1>$ rwsetmember 8.8.x.x S*.set
S0_dns_servers_out.set
S12_dns_servers_out.set
S1_dns_servers_out.set
S2_dns_servers_out.set
S3_dns_servers_out.set
<2>$ rwsiteinfo --fields=sensor,describe-sensor \
   --sensors=S0,S1,S2,S3,S12
Sensor|Sensor-Description|
    S0|           Div0Ext|
    S1|           Div1Ext|
    S2|           Div0Int|
    S3|          Div1Int1|
   S12|           Div1svc|
```

Example 4.30: Identifying DNS Traffic Flow

Help with `rwsetmember`. For a list of command options, type `rwsetmember --help` at the command line. For more information about the `rwsetmember` command, type `man rwsetmember`.

Finding Set Membership and Intersection with `rwsettool`. The `rwsettool --intersection` option can also be used to identify infrastructure that shares traffic flows. This option intersects IP addresses that are members of two sets. Command 1 of Example 4.31 shows using `rwsettool` with `--intersection` to identify the DNS servers that both the `S0` and `S1` sensors monitor.

```
<1>$ rwsettool --intersect S0_dns_servers_out.set \
   S1_dns_servers_out.set | rwsetcat
8.8.4.4
8.8.8.8
67.215.0.5
128.8.10.90
128.63.2.53
192.5.5.241
192.33.4.12
192.36.148.17
192.58.128.30
192.112.36.4
192.203.230.10
192.228.79.201
193.0.14.129
198.41.0.4
199.7.83.42
202.12.27.33
```

Example 4.31: Identifying Shared DNS Monitoring

Displaying IPset Statistics and Network Structure

In addition to printing out IP addresses as described in Section 2.2.8, `rwsetcat` can perform counting and statistical reporting. These features are useful for describing the set without displaying all of the IP addresses in the set. Since sets can have any number of addresses, counting with `rwsetcat` tends to be much faster than counting via text tools such as `wc`.

Example 4.32 illustrates how to display statistics with `rwsetcat`.

- Command 1 shows how many IP addresses are in the IPset.
- Command 2 shows summary statistics and network structure for the addresses in the IPset.

Example 4.32 also shows the wide variety of network information that can be displayed by using the `--network-structure` parameter.

- In Command 3, there are no *list-lengths* and no *summary-lengths*. As a result, a default array of summary lengths is supplied.

- In Command 4 there is a *list-length*, but no slash (`/`) introducing *summary-lengths*, so the netblock with the specified prefix length is listed, but no summary is produced.
- In Command 5, a prefix length is supplied that is sufficiently large to list multiple netblocks.
- Command 6 shows two prefix lengths in *list-lengths*.
- Command 7 shows that a prefix length of zero (no network bits, so no choice of networks) treats the entire address space as a single network and labels it `TOTAL`.
- Command 8 shows that summarization occurs not only for the *summary-lengths* but also for every prefix length in *list-lengths* that is larger than the current list length.
- In Command 9, the slash introduces *summary-lengths*, but the array of summary lengths is empty; as a result, the word "hosts" appears as if there will be summaries, but there aren't any.
- In Command 10, the `S` replaces the slash and summary lengths, so default summary lengths are used.
- In Command 11, the list length is larger than the smallest default summary length, so that summary length does not appear.
- In Command 12, `H` (host) is used for a list length.
- Command 13 shows that `H` is equivalent to 32 for IPv4.

```
<1>$ rwsetcat medtcp-dest.set --count-ips
93
<2>$ rwsetcat medtcp-dest.set --print-statistics
Network Summary
        minimumIP =       10.0.40.20
        maximumIP =  192.168.166.233
                93 hosts (/32s),    0.000002% of 2^32
                 2 occupied /8s,    0.781250% of 2^8
                 2 occupied /16s,   0.003052% of 2^16
                20 occupied /24s,   0.000119% of 2^24
                66 occupied /27s,   0.000049% of 2^27
<3>$ rwsetcat medtcp-dest.set --network-structure
TOTAL| 93 hosts in 2 /8s, 2 /16s, 20 /24s, and 66 /27s
<4>$ rwsetcat medtcp-dest.set --network-structure=4
        0.0.0.0/4| 10
      192.0.0.0/4| 83
<5>$ rwsetcat medtcp-dest.set --network-structure=18
       10.0.0.0/18| 10
    192.168.0.0/18| 15
   192.168.64.0/18| 27
  192.168.128.0/18| 41
<6>$ rwsetcat medtcp-dest.set --network-structure=4,18
  10.0.0.0/18       | 10
0.0.0.0/4           | 10
  192.168.0.0/18    | 15
  192.168.64.0/18   | 27
  192.168.128.0/18  | 41
192.0.0.0/4         | 83
<7>$ rwsetcat medtcp-dest.set --network-structure=0,18
  10.0.0.0/18       | 10
  192.168.0.0/18    | 15
  192.168.64.0/18   | 27
  192.168.128.0/18  | 41
TOTAL               | 93
<8>$ rwsetcat medtcp-dest.set --network-structure=4,18/24
  10.0.0.0/18       | 10 hosts in 2 /24s
0.0.0.0/4           | 10 hosts in 1 /18 and 2 /24s
  192.168.0.0/18    | 15 hosts in 3 /24s
  192.168.64.0/18   | 27 hosts in 6 /24s
  192.168.128.0/18  | 41 hosts in 9 /24s
192.0.0.0/4         | 83 hosts in 3 /18s and 18 /24s
<9>$ rwsetcat medtcp-dest.set --network-structure=4/
        0.0.0.0/4| 10 hosts
      192.0.0.0/4| 83 hosts
<10>$ rwsetcat medtcp-dest.set --network-structure=4S
        0.0.0.0/4| 10 hosts in 1 /8, 1 /16, 2 /24s, and 4 /27s
      192.0.0.0/4| 83 hosts in 1 /8, 1 /16, 18 /24s, and 62 /27s
<11>$ rwsetcat medtcp-dest.set --network-structure=12S
       10.0.0.0/12| 10 hosts in 1 /16, 2 /24s, and 4 /27s
    192.160.0.0/12| 83 hosts in 1 /16, 18 /24s, and 62 /27s
<12>$ rwsetcat medtcp-dest.set --network-structure=H \
| head -n 5
     10.0.40.20|
```

```
     10.0.40.23|
     10.0.40.53|
     10.0.40.83|
     10.0.50.11|
<13>$ rwsetcat medtcp-dest.set --network-structure=32 \
| head -n 5
     10.0.40.20|
     10.0.40.23|
     10.0.40.53|
     10.0.40.83|
     10.0.50.11|
```

Example 4.32: `rwsetcat` Options for Showing Structure

4.2.8 Indicating Flow Relationships

A useful step in a multi-path analysis is to identify a set of flow records that have common attributes—for instance, records that are part of the same TCP session. Use `rwgroup` and `rwmatch` to label a set of flow records that share attributes. This identifier, or group ID, is stored in the next-hop IP (`nhIP`) field. It can be manipulated as an IP address (that is, either by directly specifying a group ID or by using IPsets). The two tools generate group IDs in different ways.

- `rwgroup` scans a file of flow records and groups records with common attributes, such as source or destination IP address pairs.
- `rwmatch` groups records of different types (typically, incoming and outgoing types), creating a file containing groups that represent TCP sessions or groups that represent other behavior.

Hint 4.5: Scale Grouping Tools with Sorted Data

To improve scalability, the grouping tools require the data they process to first be sorted using `rwsort`. The sorted data must be sorted on the criteria fields: in the case of `rwgroup`, the ID field and delta fields; in the case of `rwmatch`, start time and the fields specified in the `--relate` parameter(s).

Labeling Flow Records Based on Common Attributes

The `rwgroup` command groups flow records that have common field values. Grouped records can be output separately (with each record in the group having a common ID) or summarized by a single record. Applications of `rwgroup` include the following:

- grouping together all flow records for a long-lived session: By specifying that records are grouped together by their port numbers and IP addresses, an analyst can assign a common ID to all of the flow records that make up a long-lived session.
- reconstructing web sessions: Due to diversified hosting and caching services such as Akamai®, a single webpage on a commercial website is usually hosted on multiple servers. For example, the images may be on one server, the HTML text on a second server, advertising images on a third server, and multimedia

on a fourth server. An analyst can use `rwgroup` to tag web traffic flow records from a single user that are closely related in time and then use that information to identify individual webpage fetches.

- counting conversations: An analyst can group all the communications between two IP addresses together and see how much data was transferred between both sites regardless of port numbers. This is particularly useful when one site is using a large number of ephemeral ports.

Flow records are grouped when they have matching fields. The fields specified by `--id-fields` must be identical and the field specified by `--delta-field` must match within a value less than or equal to the value specified by `--delta-value`.

Creating a new group is a two-step process:

1. Sort the records using `rwsort` as described in Section 2.2.7. `rwgroup` requires input records to be sorted by the fields specified in `--id-fields` and `--delta-field`.
2. Run `rwgroup` to create a group that matches the grouping criteria specified by `--id-fields`, `--delta-field`,and `--delta-value`. Records in the same group are assigned a common group ID.

`rwgroup` outputs a stream of flow records. Each record's next-hop IP address field is set to the value of the group ID.

Grouping Records By Session. The most basic use of `rwgroup` is to group together flow records that constitute a single longer session, such as the components of a single FTP session (or, in the case of Example 4.33, a Microsoft® Distributed File System Replication Service session). To do this, the example does the following:

1. Command 1 uses `rwfilter` to pull the desired flow records from the repository. It then uses the `rwsort` command to sort these records by source and destination IP address, source and destination port, and start time.
2. Command 2 uses `rwgroup` to group together flow records that have closely-related start times.
3. Command 3 uses the `rwfilter` and `rwcut` commands to display grouped records. It filters for records with `--next-hop-id=0.0.0.1,28` (the group IDs), then displays the source and destination IP addresses, source and destination ports, and the group ID (nhIP).

This creates a group of records that comprise a session.

Thresholding Groups By Number of Records. By default, `rwgroup` outputs one flow record for every flow record it receives as input. You can set a threshold for flow record output by using the `--rec-threshold` parameter, as shown in Example 4.34. This parameter specifies that `rwgroup` only passes records that belong to a group with at least as many records as given in `--rec-threshold`. All other records are dropped silently.

This allows you to filter out smaller groups of records. For instance, if you are only interested in grouping significant amounts of traffic, you could drop groups with low flow counts. Example 4.34 shows how this thresholding works. In the first case, there are several low-flow-count groups. When `rwgroup` is invoked with `--rec-threshold=4`, these groups are discarded by `rwgroup`, while the groups with 4 or more flow records are output.

```
<1>$ rwfilter --type=in,out --start-date=2015/06/02 \
   --end-date=2015/06/18 --packets=4- --protocol=6 \
   --bytes-per-packet=60- --duration=1000- --pass=stdout \
| rwsort --fields=1,2,3,4,sTime --output-path=sorted.rw
<2>$ rwgroup sorted.rw --id-fields=1,2,3,4 --delta-field=sTime \
   --delta-value=3600 --output-path=grouped.rw
<3>$ rwfilter grouped.rw --next-hop-id=0.0.0.1,28 --pass=stdout \
| rwcut --fields=1-4,nhIP
            sIP|            dIP|sPort|dPort|           nhIP|
     10.0.40.20|  192.168.40.20|55425| 5722|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|        0.0.0.1|
  192.168.40.20|     10.0.40.20| 5722|55425|       0.0.0.28|
  192.168.40.20|     10.0.40.20| 5722|55425|       0.0.0.28|
  192.168.40.20|     10.0.40.20| 5722|55425|       0.0.0.28|
```

Example 4.33: Grouping Flows of a Long Session with `rwgroup`

```
<1>$ rwgroup sorted.rw --id-fields=1,2,3,4 --delta-field=sTime \
   --delta-value=3600 \
| rwcut --num-recs=10 --field=1-5,nhIP
            sIP|            dIP|sPort|dPort|pro|           nhIP|
     10.0.40.20|  192.168.40.20| 5722|60309|  6|        0.0.0.0|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|        0.0.0.1|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
<2>$ rwgroup sorted.rw --id-fields=1,2,3,4 --delta-field=sTime \
   --delta-value=3600 --rec-threshold=4 \
| rwcut --num-recs=10 --field=1-5,nhIP
            sIP|            dIP|sPort|dPort|pro|           nhIP|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|        0.0.0.2|
```

Example 4.34: Dropping Trivial Groups with `rwgroup --rec-threshold`

Generating a Single Summary Record for a Group. `rwgroup` can also generate a single summary record with the `--summarize` parameter. When this parameter is used, `rwgroup` only produces a single record for each group. The summary record uses the first record in the group for its addressing information (IP addresses, ports, and protocol). The total number of bytes and packets for the group is recorded in the summary record's corresponding fields, and the start and end times for the record will be the extrema for that group.[13]

Example 4.35 shows how summarizing works: The 10 original records are reduced to two group summaries, and the byte totals for those records are equal to the sum of the byte values of all the records in the group.

```
<1>$ rwgroup sorted.rw --id-fields=1,2,3,4 --delta-field=sTime \
   --delta-value=3600 --rec-threshold=3 \
| rwcut --fields=1-5,bytes,nhIP --num-recs=10
            sIP|            dIP|sPort|dPort|pro|     bytes|           nhIP|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|      5523|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|     21084|        0.0.0.1|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|     11500|        0.0.0.1|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    977689|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|    959308|        0.0.0.2|
<2>$ rwgroup sorted.rw --id-fields=1,2,3,4 --delta-field=sTime \
   --delta-value=3600 --rec-threshold=3 --summarize \
| rwcut --fields=1-5,bytes,nhIP --num-recs=5
            sIP|            dIP|sPort|dPort|pro|     bytes|           nhIP|
     10.0.40.20|  192.168.40.20|55425| 5722|  6|     38107|        0.0.0.1|
     10.0.50.12| 192.168.40.100| 3088| 8005|  6|  46529266|        0.0.0.2|
     10.0.50.12| 192.168.40.100| 3089| 8005|  6|  46726003|        0.0.0.3|
     10.0.50.12| 192.168.40.100| 3090| 8005|  6|  46497279|        0.0.0.4|
     10.0.50.12| 192.168.40.100| 3091| 8005|  6|  46448588|        0.0.0.5|
```

Example 4.35: Summarizing Groups with `rwgroup --summarize`

Grouping Records via IPsets. For any data file, calling `rwgroup` with the same `--id-fields` and `--delta-field` values will result in the same group IDs being assigned to the same records. As a result, an analyst can use `rwgroup` to manipulate groups of flow records where the group has a specific attribute. This can be done by using `rwgroup` and IPsets, as shown in Example 4.36.

1. Command 1 uses `rwfilter` to filter for traffic with the TCP protocol (`--protocol=6`) and destination ports 20 and 21 (`--dport=20,21`). It then calls `rwsort` to sorts the data and uses `rwgroup` to convert the results into a file, `out.rw`, grouped as FTP communications between two sites. All TCP port 20 and 21 communications between two sites are part of the same group.

2. Command 2 filters through the collection of groups for those group IDs (as next-hop IP addresses stored in `control.set`) that use FTP control.

[13] This is only a quick version of condensing long flows—TCP flags, termination conditions, and application labeling may not be properly reflected in the output.

3. Finally, Command 3 uses the next-hop IPset to pull out all the flows (ports 20 and 21) in groups that had FTP control (port 21) flows.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --protocol=6 --type=out --dport=20,21 --pass=stdout \
| rwsort --fields=1,2,sTime \
| rwgroup --id-fields=1,2 --output-path=out.rw
<2>$ rwfilter out.rw --dport=21 --pass=stdout \
| rwset --nhip-file=control.set
<3>$ rwfilter out.rw --nhipset=control.set --pass=stdout \
| rwcut --fields=1-5,sTime --num-recs=5
            sIP|            dIP|sPort|dPort|pro|                  sTime|
  192.168.70.10|     10.0.40.83|65360|   21|  6|2015/06/16T20:38:57.018|
  192.168.70.10|     10.0.40.83|65360|   21|  6|2015/06/16T20:38:57.018|
  192.168.70.10|     10.0.40.83|65360|   21|  6|2015/06/16T20:38:57.146|
  192.168.70.10|     10.0.40.83|57096|   21|  6|2015/06/17T04:41:35.525|
  192.168.70.10|     10.0.40.83|57096|   21|  6|2015/06/17T04:41:35.525|
```

Example 4.36: Using `rwgroup` to Identify Specific Sessions

Help with `rwgroup`. For a list of command options, type `rwgroup --help` at the command line. For more information about the `rwgroup` command, type `man rwgroup`.

Labeling Matched Groups of Flow Records

As part of a larger analysis, you may want to group network flow records. For instance, you could group records from both sides of a bidirectional session, such as HTTP requests and responses, for further examination. You may also wish to create groups with more flexible matching, such as matching groups across protocols to identify `traceroute` messages, which use the UDP and ICMP protocols.

Use the `rwmatch` to create *matched groups*. A matched group consists of an initial record (usually a *query*) followed by one or more *responses* and (optionally) additional queries.

- A response is a record that is related to the query (as specified in the `rwmatch` command). However, it is collected from a different direction or from a different router. As a result, the fields relating the two records may be different. For example, the source IP address in one record may match the destination IP address in another record.

- A relationship in `rwmatch` is established using the `--relate` parameter, which takes two numeric field IDs separated by a comma (e.g., `--relate=3,4` or `--relate=5,5`). The first value corresponds to the field ID in the query file,. The second value corresponds to the field ID in the response file. For example, `--relate=1,2` states that the source IP address in the query file must match the destination IP address in the response file. The `rwmatch` tool will process multiple relationships, but each field in the query file can be related to, at most, one field in the response file.

Sorting Input Files Before Matching. The two input files to `rwmatch` must be sorted before matching. The same information provided in the `--relate` parameters, plus `sTime`, must be used for sorting. The first fields in the `--relate` value pairs, plus `sTime`, constitute the sort fields for the query file. The second fields in the `--relate` value pairs, plus `sTime`, constitute the sort fields for the response file.

Incomplete Query-Response Relationships. The `--relate` parameter always specifies a relationship from the query to the responses, so specifying `--relate=1,2` means that the records match if the source IP address in the query record matches the destination IP address in the response. Consequently, when working with a protocol where there are implicit relationships between the queries and responses, especially TCP, these relationships must be fully specified.

Example 4.37 shows the impact that not specifying all of the fields has on TCP data. Note that the match relationship specified (the source IP address of the query matches the destination IP address of the response) results in all of the records in the response matching the initial query record, even though the source ports in the query file may differ from a response's destination port (as seen with the third matched record).

```
<1>$ rwfilter --type=in,out --start-date=2015/06/02 \
   --end-date=2015/06/18 --protocol=6 --dport=88 \
   --pass=stdout \
| rwsort --fields=1 --output-path=query.rw
<2>$ rwfilter --type=in,out --start-date=2015/06/02 \
   --end-date=2015/06/18 --protocol=6 --sport=88 \
   --pass=stdout \
| rwsort --fields=2 --output-path=response.rw
<3>$ rwcut query.rw --fields=1-4,sTime --num-recs=4
            sIP|            dIP|sPort|dPort|                  sTime|
    10.0.40.26|  192.168.40.20|57288|   88|2015/06/17T17:06:38.871|
    10.0.40.26|  192.168.40.20|57287|   88|2015/06/17T17:06:38.865|
    10.0.40.26|  192.168.40.20|52176|   88|2015/06/16T16:22:08.659|
    10.0.40.26|  192.168.40.20|57287|   88|2015/06/17T17:06:38.856|
<4>$ rwcut response.rw --fields=1-4,sTime --num-recs=4
            sIP|            dIP|sPort|dPort|                  sTime|
  192.168.40.20|     10.0.40.26|   88|57390|2015/06/17T17:32:21.501|
  192.168.40.20|     10.0.40.26|   88|52724|2015/06/16T18:54:21.810|
  192.168.40.20|     10.0.40.26|   88|52725|2015/06/16T18:54:21.818|
  192.168.40.20|     10.0.40.26|   88|57389|2015/06/17T17:32:21.493|
<5>$ rwmatch --relate=1,2 query.rw response.rw stdout \
| rwcut --fields=1-4,nhIP --num-recs=10
            sIP|            dIP|sPort|dPort|           nhIP|
    10.0.40.26|  192.168.40.20|57390|   88|        0.0.0.1|
  192.168.40.20|     10.0.40.26|   88|57390|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|52724|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|52725|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|57389|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|51466|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|57321|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|57320|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|52030|      255.0.0.1|
  192.168.40.20|     10.0.40.26|   88|52031|      255.0.0.1|
```

Example 4.37: Using `rwmatch` with Incomplete Relate Values

Matching TCP and UDP Relationships. Example 4.38 shows the relationships that could be specified when working with TCP or UDP. This example specifies a relationship between the query's source IP address and the response's destination IP address, the query's source port and the response's destination port, and the reflexive relationships between query and response.

Matching partial associations. `rwmatch` is designed to handle not just protocols where source and destination are closely associated, but also those where partial associations are significant.

```
<1>$ rwsort query.rw --fields=1,2,sTime \
   --output-path=squery.rw
<2>$ rwsort response.rw --fields=2,1,sTime \
   --output-path=sresponse.rw
<3>$ rwmatch --relate=1,2 --relate=2,1 --relate=3,4 --relate=4,3 \
   squery.rw sresponse.rw stdout \
| rwcut --fields=1-4,nhIP --num-recs=5
            sIP|            dIP|sPort|dPort|           nhIP|
     10.0.40.26|  192.168.40.20|52396|   88|        0.0.0.1|
  192.168.40.20|     10.0.40.26|   88|52396|      255.0.0.1|
     10.0.40.26|  192.168.40.20|51466|   88|        0.0.0.2|
  192.168.40.20|     10.0.40.26|   88|51466|      255.0.0.2|
     10.0.40.26|  192.168.40.20|51466|   88|        0.0.0.3|
```

Example 4.38: Using `rwmatch` with Full TCP Fields

To establish a match group, a response record must first match a query record. For that to happen all the fields of the query record specified as first values in relate pairs must match all the fields of the response record specified as second values in the relate pairs. Additionally, the start time of the response record must fall in the interval between the query record's start time and its end time extended by the value of `--time-delta`.

Alternatively, if the `--symmetric-delta` parameter is specified, the query record's start time may fall in the interval between the response record's start time and its end time extended by `--time-delta`. The record whose start time is earlier becomes the base record for further matching.

Additional target records from either file may be added to the match group. If the base and target records come from different files, field matching is performed with the two fields specified for each relate pair. If the base and target records come from the same file, field matching is done with the same field for both records.

In addition to matching the relate pairs, the target record's start time must fall within a time window beginning at the start time of the base record. If `--absolute-delta` is specified, the window ends at the base record's end time extended by `--time-delta`. If `--relative-delta` is specified, the window ends `--time-delta` seconds after the maximum end time of any record in the match group so far. If `--infinite-delta` is specified, time matching is not performed.

Use of Next-hop IP Address in `rwmatch`. As with `rwgroup`, `rwmatch` sets the next-hop IP address field to an identifier common to all related flow records. However, `rwmatch` groups records from two distinct files into single groups. To indicate the origin of a record, `rwmatch` uses different values in the next-hop IP address field. Query records will have an IPv4 address where the first octet is set to zero; in response records, the first octet is set to 255. `rwmatch` *only* outputs queries that have a response grouped with all the responses to that query.

Discarding match queries. `rwmatch` discards queries that do not have a response and responses that do not have a query unless `--unmatched` is specified. It tells `rwmatch` to write unmatched query and/or response records to an output file or standard output.

- `q` writes query records. Unmatched query records have their next hop IPset to 0.0.0.0.
- `r` writes response records. Unmatched response records have their next hop IPset to 255.0.0.0.
- `b` writes both query and response records.

As a result, the output from `rwmatch` usually contains fewer records than the total of the two source files. `rwgroup` can be used to compensate for this by merging all of the query records for a single session into one record.

Sorting `rwmatch` input by start time. Example 4.38 matches all addresses and ports in both directions. As with `rwgroup`, `rwmatch` requires sorted data. For `rwmatch`, there is always an implicit time-based relationship controlled using the `--time-delta` parameter. As a consequence, *always* sort `rwmatch` data on the start time. (Example 4.37 generated the query and response files from a query that might not produce sorted records; Example 4.38 corrected this by sorting the input files before calling `rwmatch`.)

Help with `rwmatch`. For a list of command options, type `rwmatch --help` at the command line. For more information about the `rwmatch` command, type `man rwmatch`.

4.2.9 Managing IPset, Bag, and Prefix Map Files

During a multi-path analysis, an analyst typically performs many intermediate steps while isolating the behavior of interest. The `rwfileinfo` command prints information about binary SiLK files, helping you to manage the files generated as part of this process.

`rwfileinfo` can display information about all types of binary SiLK files: flow record files, IPset files, bag files, and prefix map (or pmap) files. Section 2.2.4 describes how to use `rwfileinfo` for viewing information about flow record files. The current section describes how to use `rwfileinfo` to view information about set, bag, and pmap files, such as the record count, file size, and the SiLK command(s) that created the file. Additionally, `rwfileinfo` displays information specific to each type of file. This information is displayed by default along with the rest of the file information, or can be explicitly viewed by using the `--fields` parameter.

- For IPset files, it shows the nodes, branches and leaves of the IPset (`--fields=ipset`).
- For bag files, it shows the key-value pairs that make up the bag (`--fields=bag`).
- For pmap files, it shows the internal prefix map name (`--fields=prefix-map`). If a prefix map was created without a map name, `rwfileinfo` returns an empty result for the prefix-map-specific field.

Example 4.39 show how `rwfileinfo` handles these files.

1. Commands 1, 2 and 3 create a set, a bag, and a pmap for the example.
2. Command 4 shows a full `rwfileinfo` result for the set file.
3. Commands 5, 6 and 7 show just the specific information for the IPset, bag, and pmap file, respectively.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --sensors=S0,S1 --type=out,outweb --protocol=0- \
   --pass=stdout \
| rwbag --bag-file=sIPv4,flows,internal.bag
<2>$ rwbagtool --coverset --ipset-record-version=4 \
   --output-path=internal.set internal.bag
<3>$ rwpmapbuild --input-file=internal.pmap.txt \
   --output-file=internal.pmap
<4>$ rwfileinfo internal.set
internal.set:
  format(id)          FT_IPSET(0x1d)
  version             16
  byte-order          littleEndian
  compression(id)     lzo1x(2)
  header-length       56
  record-length       1
  record-version      4
  silk-version        3.16.1
  count-records       1184
  file-size           506
  ipset               IPv4
<5>$ rwfileinfo internal.set --fields=ipset
internal.set:
  ipset               IPv4
<6>$ rwfileinfo internal.bag --fields=bag
internal.bag:
  bag                 key: sIPv4 @ 4 octets; counter: records @ 8 octets
<7>$ rwfileinfo internal.pmap --fields=prefix-map
internal.pmap:
  prefix-map          v1: internal
```

Example 4.39: `rwfileinfo` for Sets, Bags, and Prefix Maps

4.3 Multi-path Analysis for Situational Awareness

Analysts support situational awareness for cybersecurity by determining what is present, what is threatening, and what potential developments (either aggressive or defensive) could be expected. Multi-path analyses often focus on the "what is threatening" part by comparing traffic samples that have been collected under varying conditions.[14] In other words, they explore *differential awareness*—inferring when "what should be" does not match "what is." (see Section 2.3.1.)

By investigating the behavior of the network under different conditions, analysts can figure out the difference between normal and abnormal activity. Examples of different conditions that might be examined during a multi-path analysis include samples of data before and after a specific event (such as an indicator of compromise), samples of traffic from both expected and unexpected external addresses or ports, and samples of normal and abnormal sizes and durations. Analysts can then use this information to infer what might be happening on the network. Their ultimate goal is to determine whether this activity presents a threat to the organization.

4.3.1 Structuring a Multi-path Analytic for Situational Awareness

A multi-path analysis for situational awareness incorporates an analytic that retrieves multiple pools of traffic that differ in the selected condition. (In contrast, a single-path analysis often will pull just one set of data and then refine or divide it.) Each data pool is then examined separately and processed to produce an internal state that can be compared to identify or indicate threatening behavior.

Selecting this internal state is a key step in a multi-path analysis of situational awareness. Analysts often use some form of count indexed by a characteristic. The type of count they select depends on the focus of the analysis.

- Count flows if the analysis focuses on frequency of behavior.
- Count packets if the focus is on the volume of behavior.
- Count bytes if the focus is on bandwidth consumption.

The indexing characteristic is driven by the condition that forms the branches in the analysis.

- For service-based analyses, index with transport protocol or port.
- For event-based analyses, index by time period.
- For vulnerability-based analyses, index by a volumetric such as duration or size.

Once the values (indexed counts) in these data sets are calculated, you can analyze differences or contrasts between them to provide differential awareness.

[14] In contrast, single-path analyses such as those discussed in Section 2.3 often focus on the "what is present" part of situational awareness.

4.3.2 Characterizing Threats

Situational awareness complements threat hunting. You need to understand your network's behavior to identify changes that might indicate threats. Network behavior is not static; awareness of how it shifts over time is key to finding threats. To provide context for threat identification, first retrieve and process additional datasets. Then, analyze each dataset to gain insight into potential threats.

- If the threat is related to a specific event, pull data by time intervals before, during, and after the event. Profile this data by computing descriptive statistics, such as number of communicating endpoints, bytes/flow average, and bytes/packet average. Comparing these statistics can show the impact of the event and assist in finding characteristics that either identify the threat or determine which direction to anticipate activity by the threat actors.
- If the threat is indicated by specific endpoints (e.g., source or destination IP addresses), pull data by the indicative addresses, by other addresses contacted by the indicative addresses, and by unrelated addresses using similar ports and protocols. Generate time series and frequency counts for each pool of data, then compare the results to indicate the threat.
- If the threat is indicated by counts of bytes or packets (either unusual in isolation or unusually similar in magnitude), pull data by counts within the expected ranges, outside the expected ranges but not threatening, and within the threatening range. Profile each of these data pools using a time series (with counts indexed by endpoints), then compare them to indicate the threat.
- If the threat is indicated by network port usage (either on the characteristic port or off of it), pull the data according to whether the port is used as the source, the destination, or neither (where application recognition is used rather than the port). Profiling each of these data pools using a time series or descriptive statistics can highlight differences that indicate the threat.
- If the threat is associated with workday or after-hours traffic, pull the data by date-time groups to separate these time intervals. Profiling each data pool by communication with external addresses or by descriptive statistics may provide a useful comparison to indicate the threat.
- If the threat is associated with the frequency of communication on a given service (instead of the volume of communication), pull the data according to external endpoints that are associated with high, moderate, and low frequency of communication on that service. Profiling each pool with descriptive statistics may provide a useful comparison to indicate the threat.

4.3.3 Profiling Data

To profile the different data pools, use the SiLK tools that generate summary values.

- Use `rwcount` to generate time series values, which can be contrasted using matrix difference to highlight divergences.
- Use `rwstats --overall-stats` to compute distribution information, including minimum, quartiles, median, maximum, and histograms. These statistics are generated for bytes/flow, packets/flow, and bytes/packet, calculated over all of the flows. Example 4.40 shows a sample output for a pool of traffic.

- Use `rwstats --detail-proto-stats` to compute similar statistics to `--overall-stats`, but on a protocol-by-protocol basis. This parameter takes a list of protocol numbers as its argument. Then, use Linux text-processing commands on the `rwstats` output to isolate values of interest in profiling traffic.
- Use the `rwbag` or `rwbagbuild` command to store useful keys and values in a bag. Then, use `rwbagcat --print-statistics` to generate a statistical summary of the values in the bag. Example 4.41 shows a sample output for the pool of traffic.

When applying these tools, the general approach is to first produce a profile of each individual data pool in parallel. Then, contrast all of the profiles to identify threats of interest. This approach avoids inefficient looping between the data pools while trying to interpret each in the context of all others. Instead, these profiles should highlight the characteristic differences directly, making it easier to identify the threat.

4.3.4 Multi-path Analysis for Differential Awareness: Investigate Abnormal Web Traffic

Let's continue the sample situational awareness analysis from Section 2.3.2. Our analyst Arthur previously investigated the behavior of his organization's web and DNS servers to validate that they were functioning properly and find any unusual behavior (desired awareness). He also looked as the traffic carried by the web and DNS servers to see if it deviated from expected patterns (actual awareness). Arthur identified a host that was behaving anomalously and found an unusual traffic pattern on the web server.

Arthur now shifts his focus from investigating expected web traffic to investigating abnormal or threatening traffic. He will profile web server activity on different ports to see if he can find more evidence of abnormal behavior that might indicate potentially hostile activity. In other words, he will perform a differential awareness analysis of web traffic.

Profile On-port and Off-port Web Traffic

Arthur's investigation starts by looking at recognized web traffic. Example 4.42 retrieves outbound traffic (`out` and `outweb`) by application labeling of 80 or 443, and examines both on-port activity (i.e., traffic on normal web ports) and off-port activity (i.e., traffic without normal web ports).

Command 1 uses `rwfilter` and `rwstats` to profile the on-port traffic. The results show the top 5 destination ports. As Arthur expected, normal web ports (80, 443, 8080, and 8443) account for more than 98% of the flows, with the remainder being dynamic ports.

Command 2 similarly uses `rwfilter` and `rwstats` to profile off-port traffic. The source ports are used in this command, as the destination ports were dominated by dynamic ports.

- The first two ports (591 and 8008) are alternative HTTP ports.
- The next two ports (8005 and 8834) are associated with streaming web services.
- The last port (139) is a Windows NetBIOS service port with known security issues. This service is not normally configured to us HTTP or TLS, and so its presence on the profile is doubly suspicious.

```
<1>$ rwstats --overall-stats traffic.rw
FLOW STATISTICS--ALL PROTOCOLS:  20840493 records
*BYTES min 23; max 68282013
  quartiles LQ 86.15025 Med 141.36326 UQ 285.97591 UQ-LQ 199.82566
   interval_max|count<=max|%_of_input|   cumul_%|
             40|    316815|  1.520190|  1.520190|
             60|    451856|  2.168164|  3.688353|
            100|   6793743| 32.598763| 36.287117|
            150|   3454554| 16.576163| 52.863279|
            256|   4508177| 21.631816| 74.495095|
           1000|   2611652| 12.531623| 87.026718|
          10000|   2637017| 12.653333| 99.680051|
         100000|     60954|  0.292479| 99.972529|
        1000000|      4683|  0.022471| 99.995000|
     4294967295|      1042|  0.005000|100.000000|
*PACKETS min 1; max 405734
  quartiles LQ 1.13593 Med 2.27186 UQ 4.94753 UQ-LQ 3.81160
   interval_max|count<=max|%_of_input|   cumul_%|
              3|  13759975| 66.025190| 66.025190|
              4|   1222465|  5.865816| 71.891006|
             10|   4102843| 19.686881| 91.577886|
             20|   1335366|  6.407555| 97.985441|
             50|    304678|  1.461952| 99.447393|
            100|     55297|  0.265334| 99.712728|
            500|     55378|  0.265723| 99.978451|
           1000|      2439|  0.011703| 99.990154|
          10000|      1752|  0.008407| 99.998560|
     4294967295|       300|  0.001440|100.000000|
*BYTES/PACKET min 21; max 6269
  quartiles LQ 61.81097 Med 77.66906 UQ 93.52715 UQ-LQ 31.71618
   interval_max|count<=max|%_of_input|   cumul_%|
             40|    650694|  3.122258|  3.122258|
             44|   2349349| 11.273001| 14.395259|
             60|   1615092|  7.749778| 22.145038|
            100|  13141869| 63.059300| 85.204338|
            200|   1270185|  6.094793| 91.299131|
            400|   1719319|  8.249896| 99.549027|
            600|     71663|  0.343864| 99.892891|
            800|     10637|  0.051040| 99.943931|
           1500|     11519|  0.055272| 99.999203|
     4294967295|       166|  0.000797|100.000000|
```

Example 4.40: Using `rwstats --overall-stats` for Summary Statistics

```
<1>$ rwbag traffic.rw --pmap-file=ports:port.pmap \
   --bag-file=dport-pmap:ports,flows,traffic.bag
<2>$ rwbagcat --pmap-file=ports:port.pmap --print-statistics \
   traffic.bag

Statistics
    number of keys:  20
   sum of counters:  20840493
       minimum key:  icmp-any
       maximum key:  udp-llmnr
   minimum counter:  21
   maximum counter:  8404597
              mean:  1.042e+06
          variance:  4.493e+12
standard deviation:  2.12e+06
              skew:  2.276
          kurtosis:  18.05
   nodes allocated:  0 (0 bytes)
   counter density:  inf%
```

Example 4.41: Using `rwbagcat --print-statistics` for Summary Statistics

```
<1>$ rwfilter --start=2015/06/17 --type=out,outweb \
   --application=80,443 --aport=80,8080,443,8443 \
   --pass=stdout \
| rwstats --fields=dport --values=flows --count=5
INPUT: 120338 Records for 1296 Bins and 120338 Total Records
OUTPUT: Top 5 Bins by Records
dPort|   Records|  %Records|   cumul_%|
   80|     67574| 56.153501| 56.153501|
  443|     47195| 39.218701| 95.372202|
 8443|      4136|  3.436986| 98.809187|
 8080|         3|  0.002493| 98.811680|
52576|         3|  0.002493| 98.814173|
<2>$ rwfilter --start=2015/06/17 --type=out,outweb \
   --application=80,443 --pass=stdout \
| rwfilter stdin --aport=80,8080,443,8443 --fail=stdout \
| rwstats --fields=sport --values=flows --count=5
INPUT: 1784 Records for 129 Bins and 1784 Total Records
OUTPUT: Top 5 Bins by Records
sPort|   Records|  %Records|   cumul_%|
  591|      1386| 77.690583| 77.690583|
 8008|       123|  6.894619| 84.585202|
 8005|        64|  3.587444| 88.172646|
 8834|        64|  3.587444| 91.760090|
  139|        17|  0.952915| 92.713004|
```

Example 4.42: Using `rwfilter` and `rwstats` for Contrasting On-port and Off-port Web Connections

Investigate Traffic on Port 139

To dig deeper into the suspicious activity on port 139, Arthur looks at identified web flows as shown in Example 4.43. In contrast to the previous example, Command 1 pulls both directions of web-based traffic involving port 139. It stores the flows in the file `off-port.rw` and also passes the flows to `rwcut` for display.

While just the first 10 flows (sorted by start time) are shown, several things are apparent:

- Flows are shown going both ways (to and from port 139).
- The source address of the traffic to port 139 is often or exclusively the host `192.168.181.8`.
- The destination addresses on which port 139 is accessed vary.
- They byte counts involved are small, particularly for web traffic (low hundreds of bytes per flow).
- The timing of the flows is too rapid for this to be browser traffic.

Arthur then uses `rwuniq` to further examine the flows saved by Command 1, as shown in Commands 2 through 5. The results of these commands confirm that `192.168.181.8` is the only source address on flows to port 139, and the only destination address on flows from port 139. Further examination of the results reveals several things:

- The TCP connections to port 139 all have the same set of flags, which further indicates that this are not browser connections.
- Each of the 17 connections to port 139 produce a response from port 139, which indicates that this is not random port access.
- All of the connections initiate and terminate normally, but with very few packets per flow, and very small (below 60) bytes per packet averages.

Conclusions from the Multi-path Analysis for Situational Awareness

Arthur's differential analysis of web traffic uncovered suspicious activity on port 139, which is not normally configured to use HTTP or TLS and has known security problems. He found traffic that uses a web protocol but is not web traffic, on a port that may offer administrative access to targeted machines.

As such, this is worth looking into in more depth (perhaps pulling packet content) to definitively find out if the traffic is malicious. Arthur will continue his analysis in Section 6.3.3.

```
<1>$ rwfilter --start=2015/06/17 --type=in,out \
   --application=80,443 --aport=139 --pass=off-out.rw \
   --pass=stdout \
| rwsort --fields=stime \
| rwcut --fields=1-4,bytes,stime --num=10
            sIP|            dIP|sPort|dPort|     bytes|                  sTime|
  192.168.121.2|  192.168.181.8|  139|43040|       164|2015/06/17T16:13:06.446|
  192.168.181.8|  192.168.121.2|43040|  139|       338|2015/06/17T16:13:06.446|
  192.168.181.8| 192.168.121.77|55546|  139|       350|2015/06/17T16:13:06.447|
  192.168.181.8| 192.168.121.57|46770|  139|       286|2015/06/17T16:13:06.447|
  192.168.181.8|192.168.121.145|60557|  139|       350|2015/06/17T16:13:06.447|
 192.168.121.57|  192.168.181.8|  139|46770|       169|2015/06/17T16:13:06.447|
192.168.121.145|  192.168.181.8|  139|60557|       169|2015/06/17T16:13:06.447|
 192.168.121.77|  192.168.181.8|  139|55546|       169|2015/06/17T16:13:06.447|
192.168.121.158|  192.168.181.8|  139|47620|       169|2015/06/17T16:13:06.448|
  192.168.181.8|192.168.121.158|47620|  139|       350|2015/06/17T16:13:06.448|
<2>$ rwfilter off-out.rw --dport=139 --pass=stdout \
| rwuniq --fields=sip \
   --values=bytes,packets,flows,distinct:dip \
            sIP|               Bytes|        Packets|   Records|dIP-Distin|
  192.168.181.8|                5862|            101|        17|        17|
<3>$ rwfilter off-out.rw --dport=139 --pass=stdout \
| rwuniq --fields=flags \
   --values=bytes,packets,flows,distinct:dip \
   flags|               Bytes|        Packets|   Records|dIP-Distin|
FS PA   |                5862|            101|        17|        17|
<4>$ rwfilter off-out.rw --sport=139 --pass=stdout \
| rwuniq --fields=dip \
   --values=bytes,packets,flows,distinct:sip \
            dIP|               Bytes|        Packets|   Records|sIP-Distin|
  192.168.181.8|                2875|             51|        17|        17|
<5>$ rwfilter off-out.rw --sport=139 --pass=stdout \
| rwuniq --fields=flags \
   --values=bytes,packets,flows,distinct:sip \
   flags|               Bytes|        Packets|   Records|sIP-Distin|
FS PA   |                2547|             45|        15|        15|
FS  A   |                 328|              6|         2|         2|
```

Example 4.43: Using `rwcut` and `rwuniq` to Examine Off-port Web Connections

4.4 Summary of SiLK Commands in Chapter 4

Command	Section Name	Page
`rwfilter`	Complex Filtering With `rwfilter`	75
	Finding Low-Packet Flows with `rwfilter`	80
`rwcount`	Approximating Flow Behavior Over Time	81
`rwuniq`	Using Thresholds to Profile a Slice of Flows	82
	Profiling With Compound Keys	85
	Isolating Behaviors of Interest	87
`rwstats`	Profiling With Compound Keys	85
`rwbag`	Generating Bags from Network Flow Data	89
`rwbagbuild`, `rwscan`	Generating Bags From IP Sets or Text	90
`rwbagtool`	Comparing the Contents of Bags	93
	Extracting IPsets from Bags	95
	Intersecting Bags and IPsets	95
`rwnetmask`	Masking IP Addresses	95
`rwsetbuild`	Creating IPsets from Text Files	97
`rwsetmember`	Using Set Membership to Understand Traffic Flow	99
`rwsetcat`	Displaying IPset Statistics and Network Structure	101
`rwgroup`	Labeling Flow Records Based on Common Attributes	104
`rwmatch`	Labeling Matched Groups of Flow Records	108
`rwfileinfo`	Managing IPset, Bag, and Prefix Map Files	111

Chapter 5

Case Studies: Intermediate Multi-path Analysis

This chapter features detailed case studies of multi-path analyses, using concepts from previous chapters. They employ the SiLK workflow, `rwfilter` and other SiLK tools, UNIX commands, and networking concepts to provide practical examples of multi-path analyses with network flow data.

Upon completion of this chapter you will be able to

- describe how to combine several tasks to form a multi-path analyses
- execute multi-path analyses with various SiLK tools in one automated program
- associate sets of IP addresses (IPsets) with network flow sensors

These case studies use the FCCX dataset described in Section 1.8.

5.1 Building Inventories of Network Flow Sensors With IPsets

Flow sensors commonly monitor strategic points in enterprise networks where different network environments meet. This environmental complexity affects sensor flow collection and analyst knowledge as network infrastructure evolves. For example, multiple sensors may overlap their flow collection for failover purposes; as the network routes traffic, analysts may need to determine which sensor is the primary flow collector.

Naomi has been asked to create and maintain an up-to-date inventory of network sensors. Her organization's network is constantly in flux and her team needs to know how their sensors function in this changing environment. Her inventory will help her colleagues to better understand how their sensors monitor the network, simplify the process of reviewing and validating sensors, and mitigate the issues mentioned above.

To create a sensor inventory, Naomi applies the analysis workflow described in Section 1.5 to build a multi-path analysis from successive single-path analyses. Her inventory consists of SiLK IPsets that contain internal network addresses monitored by a flow sensor. She'll create the sensor inventory by merging the results of three single-path analyses into a single IPset inventory as follows:

1. Path 1 associates network addresses with a single sensor (Section 5.1.1).
2. Path 2 associates network addresses of the remaining sensors (Section 5.1.2).
3. Path 3 associates network shared addresses (Section 5.1.3).
4. Finally, the results of each part of the multi-path analysis are merged to create a complete inventory of sensors (Section 5.1.4).

The steps she takes and the SiLK commands she uses are shown in Example 5.1.

```
<1>$ rwfilter --sensor=S0 --type=out,outweb --start=2015/06/01 \
   --end=2015/06/30 --proto=0- --pass=stdout \
| rwset --sip-file=S0-out.set --copy=stdout \
| rwbag --bag-file=sipv4,flows,S0-out.bag
<2>$ rwfilter \
   --sensor=S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,\
   S16,S17,S18,S19,S20,S21 \
   --type=out,outweb --start=2015/06/01 --end=2015/06/30 \
   --proto=0- --pass=stdout \
| rwset --sip-file=S0-other.set --copy=stdout \
| rwbag --bag-file=sipv4,flows,S0-oth.bag
<3>$ rwsettool --difference S0-out.set S0-other.set \
   --output=S0-only.set
<4>$ rwbagtool --compare=ge S0-out.bag S0-oth.bag \
   --output=S0-most.bag
<5>$ rwbagtool --coverset S0-most.bag --output=S0-most.set
<6>$ rwsetcat --net=27 S0-most.set S0-only.set \
| cut -f1 -d '|' \
| rwsetbuild stdin cand.set
<7>$ rwfilter --sensor=S0 --type=in,inweb --start=2015/06/01 \
   --end=2015/06/30 --dipset=cand.set --pass=stdout \
| rwset --dip-file=S0-in.set
<8>$ rwsettool --difference S0-in.set S0-most.set S0-only.set \
   --output=S0-close.set
<9>$ rwsettool --union S0-only.set S0-most.set S0-close.set \
   --output=S0.set
```

Example 5.1: Building an IPset Inventory for Sensor S0

5.1.1 Path 1: Associate Addresses with a Single Sensor

To begin generating IPsets, Naomi selects sensor `S0` as the *inventory sensor*.

1. In Command 1 of Example 5.1, Naomi calls the `rwfilter` command to retrieve outbound traffic (`--type=out,outweb`) on inventory sensor `S0` during the defined time period.

2. She then uses a Unix pipe (`|`) to send the records that pass the `rwfilter` query to the `rwset` command. This generates an IPset (`S0-out.set`) that contains all outbound source IP addresses that are monitored by sensor `S0`.

3. The `rwset --copy` switch copies the `rwfilter` results to standard output. Naomi again uses a Unix pipe (`|`) to send the copied IPset to the `rwbag` command. This generates a SiLK bag of flow counts per source IP address for the same time period (`S0-out.bag`).

The resulting IPset and bag provide a detailed inventory of all source IP addresses and their flow volumes monitored by sensor `S0`.

5.1.2 Path 2: Associate Addresses of Remaining Sensors

Command 2 of Example 5.1 shows the next path in Naomi's multi-path analysis: associating the addresses for non-inventory sensors in the repository. This procedure is similar to the one she used in Command 1. However, Command 2 selects outbound traffic for the remaining repository sensors (`S1` through `S21`) and generates source IP address sets and bag files for the same time period (`S0-other.set`, `S0-oth.bag`).

5.1.3 Path 3: Associate Shared Addresses

The third path of Naomi's multi-path analysis associates the addresses that may be monitored (or *shared*) by multiple sensors.

Identify Uniquely-Monitored Addresses

To begin path three, Naomi finds the IP addresses that are uniquely monitored by the inventory sensor, `S0`.

1. Command 3 of Example 5.1 calls `rwsettool --difference` to generate an IPset of addresses that are uniquely monitored by sensor `S0` (`S0-only.set`). This command compares the set of addresses from Command 1 that are monitored by `S0` (`S0-out.set`) to the set of addresses from Command 2 that are monitored by all the other sensors (`S0-other.set`). It saves the IP addresses that are only found in `S0-out.set` to `S0-only.set`.

2. Command 4 checks to see if the inventory sensor `S0` is the primary flow collection sensor. It compares the IP address flow volumes of the inventory sensor to those of the other sensors to determine if `S0` monitors the majority of network traffic for an IP address. To perform this comparison, it calls `rwbagtool` to compare the bag files created in Commands 1 and 2 (`S0-out.bag` and `S0-oth.bag`). IP addresses that are monitored by the inventory sensor and have flow volumes greater than or equal to those monitored by the non-inventory sensors are saved to the *S0-most.bag* file.

3. Command 5 calls `rwbagtool --coverset` to generate an IPset (`S0-most.set`) from the `S0-most.bag` file.

Find Asymmetric and Missing Flow Data

Naomi next accounts for asymmetric and missing network flow data, which can be caused by outages, routing, and network backdoors. To address each case, she needs to identify inbound network traffic to internal hosts

that are closely adjacent to other inventory sensor addresses. She can accomplish this by using a small CIDR range, such as /27.

1. Command 6 of Example 5.1 calls the `rwsetcat` and `rwsetbuild` commands to build a `cand.set` candidate set from the `S0-only` and `S0-most` IPset files. The `--net=27` parameter specifies the CIDR range for closely adjacent addresses in the two sets.

2. Naomi then uses this candidate set as a destination IPset (`--dipset=cand.set`) for the `rwfilter` query in Command 7. This identifies any remaining addresses not previously found in Commands 1 and 2.

3. Command 8 then calls `rwsettool --difference` to create the `S0-close.set` IPset, which contains IP addresses that are closely adjacent to inventory sensor addresses.

5.1.4 Merge Address Results

After Naomi completes the third path of the multi-path analysis, she has three subsets of the sensor inventory: `S0-only.set`, `S0-most.set`, and `S0-close.set`. To complete the full sensor inventory, she must merge all three subsets into a single IPset that contains the inventory of IP addresses associated with the inventory sensor, `S0`.

Command 9 of Example 5.1 generates the final sensor inventory with the `rwsettool --union` switch. This command performs an algebraic *union* on `S0-only.set`, `S0-most.set`, and `S0-close.set` to produce `S0.set`, which contains all of the IP addresses in the sensor inventory.

Naomi and her team can now use the sensor inventory IPset to gain a better understanding of the networks monitored by a specific network flow sensor (in this example, `S0`) and identify any situational awareness changes.

5.2 Automating IPset Inventories of Network Flow Sensors

In Section 5.1, our analyst Naomi manually generated a sensor inventory by performing a multi-path analysis. This process is fine if she only needs to inventory a single sensor a single time. However, what if she wants to inventory all of the sensors in the SiLK repository? What if she needs to update the sensor inventory regularly to capture changes to her organization's network infrastructure? Manually executing this workflow over and over would be time consuming and represents an inefficient use of her time and resources.

Naomi therefore creates a shell program to automatically execute this workflow. Her automated program uses the `rwsiteinfo` command to compile a list of *all* sensors in the SiLK repository, then executes the analytic from Section 5.1 for every sensor. Automating the workflow is more efficient and accurate than manually executing it for each sensor. It makes compiling and updating a complete sensor inventory much easier: simply run the program as needed to automatically generate a new inventory for all sensors that are currently in the repository. This improves Naomi and her team's situational awareness of the network.

Naomi's automated sensor inventory is shown in Example 5.2. It's a simple Bash shell program that carries out the process of building IPset inventories for all of the sensors in the SiLK repository. Her program consists of two main sections, a *header* and *loop*, that are discussed in detail below.

5.2.1 Program Header

The first section of Naomi's automated program is the *header*, which contains information used throughout program execution.

Line 1 of Example 5.2, commonly known as the *she-bang*, identifies this as a *Bourne-again (Bash)* shell program to the system shell. The comment in Line 2 helps users to understand the overall purpose of the program.

Lines 5-8 define variables that are referenced throughout the remainder of the program.

- The `START` and `END` variables specify the dates that are used with `rwfilter --start-date` and `--end-date` switches.
- The `INBOUND` and `OUTBOUND` variables specify the options for inbound and outbound `rwfilter` queries that are executed in the program's *for loop*.

5.2.2 Program Loop

The rest of Naomi's program is a Bash *for loop* that executes most of the program's automated tasks.

Line 11 of Example 5.2 loops through the sensors in the SiLK repository. The `rwsiteinfo --sensor` command generates a complete list of sensors in the repository. The for loop cycles through this list to generate an inventory for each sensor.

Lines 13-15 define a variable (`FILES` in this example) of temporary files that will be deleted at the end of the for loop (Line 47).

Line 17 follows with a visual output of the current inventory sensor using the UNIX `echo` command. This indicates that the program has started building an IPset inventory for that sensor.

With the exception of Lines 23-25, the remainder of the program repeats the analytic in Example 5.1 using the variables defined in the program header. Lines 23-25 are an addition to the original analytic. They call `rwsiteinfo` with the `--fields=sensor:list` option to generate a comma-delimited list of sensors. This list of sensors is then passed to a series of `sed` commands to remove the inventory sensor, effectively building an `OTHERS` variable that contains a list of non-inventory sensors. The `OTHERS` variable is then used to build the set of non-inventory sensors used in the rest of the analytic.

Line 49 again displays the name of the current sensor and indicates that the program has finished building an inventory for that sensor.

```
#!/bin/bash
# Script to automate sensor IPSet inventories

# Variables
START="2015/06/01"
END="2015/06/30"
OUTBOUND="--type=out,outweb --start=$START --end=$END --proto=0-"
INBOUND="--type=in,inweb --start=$START --end=$END --dipset=cand.set"

# Loop through each sensor in the repository
for SEN in $(rwsiteinfo --no-titles --no-final --fields=sensor); do

  FILES="$SEN-out.set $SEN-out.bag $SEN-close.set $SEN-other.set"
  FILES+=" $SEN-oth.bag $SEN-only.set cand.set"
  FILES+=" $SEN-most.set $SEN-in.set $SEN-most.bag"

  echo "Sensor: $SEN -- Start"

  rwfilter --sensor=$SEN $OUTBOUND --pass=stdout \
   | rwset --sip-file=${SEN}-out.set --copy=stdout \
   | rwbag --bag-file=sipv4,flows,${SEN}-out.bag

  OTHERS=$(rwsiteinfo --no-titles --no-final --fields=sensor:list \
   | sed -e "s/,$SEN,/,/" \
   | sed -e "s/^$SEN,//" | sed -e "s/,$SEN$//")

  rwfilter --sensor=$OTHERS $OUTBOUND --pass=stdout \
   | rwset --sip-file=${SEN}-other.set --copy=stdout \
   | rwbag --bag-file=sipv4,flows,${SEN}-oth.bag

  rwsettool --difference ${SEN}-out.set ${SEN}-other.set \
   --output=${SEN}-only.set
  rwbagtool --compare=ge ${SEN}-out.bag ${SEN}-oth.bag \
   --output=${SEN}-most.bag
  rwbagtool --coverset ${SEN}-most.bag --output=${SEN}-most.set

  rwsetcat --net=27 ${SEN}-most.set ${SEN}-only.set \
    | sed -e 's/|.*//'     | rwsetbuild stdin cand.set

  rwfilter --sensor=$SEN $INBOUND --pass=stdout \
    | rwset --dip-file=${SEN}-in.set

  rwsettool --difference ${SEN}-in.set ${SEN}-most.set \
  ${SEN}-only.set --output=${SEN}-close.set
  rwsettool --union ${SEN}-only.set ${SEN}-most.set \
  ${SEN}-close.set --output=${SEN}.set
  rm -f $FILES

  echo "Sensor: $SEN -- End"
done
```

Example 5.2: Automating IPset Inventories

Chapter 6

Advanced Exploratory Analysis with SiLK: Exploring and Hunting

This chapter introduces advanced exploratory analysis through application of the analytic development process with the SiLK tool suite. It discusses the exploratory analysis process, starting points for exploration, and advanced SiLK commands and techniques that can be employed during an analysis.

Upon completion of this chapter you will be able to

- describe advanced exploratory analysis and how it maps to the analytic development process
- describe SiLK tools that often are used during exploratory analysis
- provide example workflows for exploratory network flow analysis
- describe how to apply the exploratory analysis workflow to situational awareness

6.1 Exploratory Analysis: Concepts

The *exploratory* approach is the most open-ended of the analysis workflows described in this guide. It provides a framework for investigating more complex questions about network traffic. As the name suggests, exploratory analysis involves asking questions about trends, processes, and dynamic behavior that do not necessarily have fixed or obvious answers. Often the analysis leads to more questions!

Exploratory analysis uses single-path and multi-path analyses as building blocks to provide insight into network events. It typically considers more than one indicator of network behavior to provide a more complete understanding of what happened. Each building block represents a question (or part of a question) whose answer feeds into subsequent steps in the analysis. Analysts can assemble these building blocks by hand to prototype an analysis or examine one-time phenomena, iterating if necessary. This manual analysis can then transition to scripted analysis to save time, make it easier to repeat the analysis, and ensure more consistent results.

As Figure 6.1 indicates, this form of analysis rarely proceeds directly from initial question to final result. Instead, it gathers a variety of contextual information, and goes forward in a search pattern to reach the result.

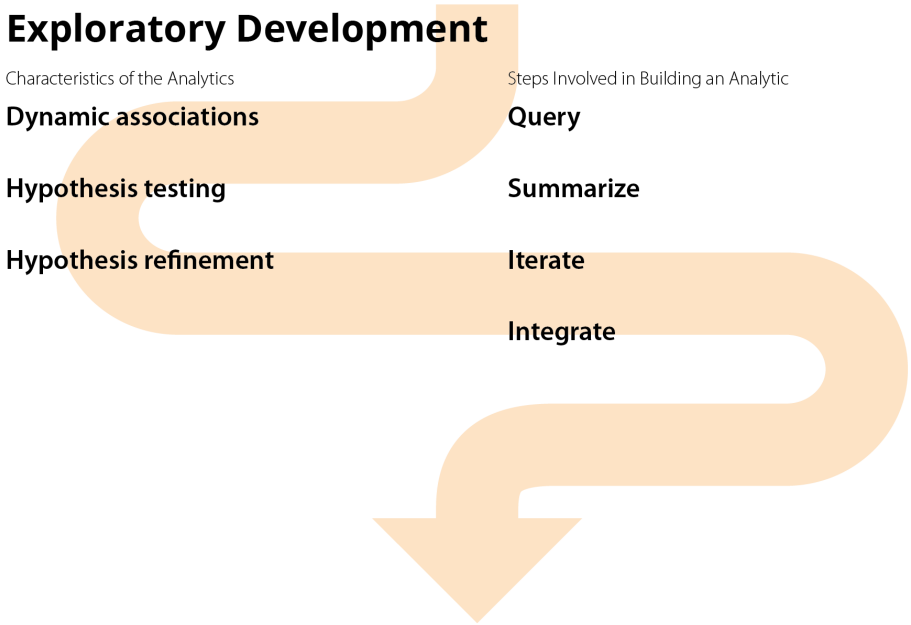

Figure 6.1: Exploratory Analysis

6.1.1 Exploring Network Behavior

Exploratory analysis follows the general SiLK analysis workflow described in Figure 1.3. The overall course of exploratory analysis generally follows these steps:

1. come up with initial questions as the starting point (the Formulate stage)
2. apply a set of analyses to provide insight into a given question (the Model and Test stages)
3. integrate the results of these analyses with previously-available data (the Analyze stage)
4. refine the results (the Refine stage)
5. identify new questions to be investigated with further analyses (return to the Formulate stage to begin another iteration of the exploratory analysis)

In contrast to single-path and multi-path analysis, analysts explore with iterations of questions and their associated analyses. We may not have a specific set of behaviors, known event, or identified service in mind at the start of an exploratory analysis. Instead, we start with a general question and seek a clearer target during the iterations of the analysis.

As the exploration progresses, the scope of these questions often becomes more associated with any features of interest that are identified in the data. These features can be identified during the initial stages of establishing context for the network behavior to be investigated and gathering data about it. They also can emerge from the results of preliminary single-path or multi-path analyses of the data, which help to refine the scope of the exploratory analysis.

Exploratory analysis often takes a deep dive into the data to examine the possible causes or conditions related to the features of interest. The analyst may form, test, and either continue to investigate or abandon hypotheses about sources of the behavior. This exploration continues until the analyst reaches a sufficient level of confidence in understanding the features to either identify the causes of the event and share the results, or close the hypothesis. When all threatening hypotheses have been closed, we can close the investigation and associate the behavior with unremarkable network activity.

6.1.2 Starting Points for Exploratory Analysis

The first question that normally arises during an exploratory analysis is, "Where should it start?"

One set of starting points for exploratory analyses involves investigating unexplained activity on the monitored network. This produces initial questions such as

- What activity is present that could be malicious or damaging?
- What is the impact of this unexpected activity?
- What led to this unexpected activity?

The initial stage of the investigation may be a relatively straightforward profiling analysis or an inventory of traffic of interest across specific servers. Further iterations use the results of these profiles and inventories to identify outliers, inconsistencies, and behaviors exhibited by the unexplained activity. These outliers, inconsistencies, and behaviors then become the focus of exploration for the next iteration. At this point in the analysis, a case-by-case frequency summary or time-based trends can provide insight into the activity. The iterations of analysis continue until the analyst identifies a reasonable explanation of the activity, rejecting alternate explanations.

Another set of starting points for exploratory analyses involves unusual levels or directions of network services. This produces initial questions such as

- What led to these services?
- Is there a change in the population of external hosts employing these services?
- Does this change in service match with a model of network attack?

Analyses for these questions often summarize network activities via time series or service-by-service counts of incoming activity versus outgoing activity. The analyst integrates and examines these summaries and counts, looking for points of significant change in trends or traffic levels. These points of change, and the hosts involved in those changes, become the focus of the next iteration of the exploratory analysis.

6.1.3 Example Exploratory Analysis: Investigating Anomalous NTP Activity

As an example of an exploratory analysis, consider an investigation into Network Time Protocol (NTP) usage. Madison is a traffic analyst who is tasked with developing a profile of the NTP server activity, particularly focused around traffic on June 16, 2015. She is looking at unusual NTP activity on her organization's network. Since NTP is associated with UDP port 123, her starting point would be to find out what level of activity is observed on this port.

Compare Inbound and Outbound UDP Activity on Port 123

The first step in Madison's analysis delves into inbound and outbound activity on port 123. The `rwfilter` commands in Example 6.1 provide an initial view of UDP activity on this port. Command 1 profiles the outgoing activity to UDP/123. Command 2 profiles the incoming activity from this port.

The size of the outbound activity is surprisingly large: almost double the number of flow records than the inbound activity, and one third larger in bytes. Madison decides that it is worth a closer look.

```
<1>$ rwfilter --start=2015/06/16 --sensor=S0,S1,S2,S3,S4 \
   --type=out --proto=17 --dport=123 --pass=NTP-out.raw \
   --print-vol=stdout
     |               Recs|            Packets|                Bytes|      Files|
Total|            5566807|           15791472|           2646290347|         60|
 Pass|               7040|               8410|               659800|           |
 Fail|            5559767|           15783062|           2645630547|           |
<2>$ rwfilter --start=2015/06/16 --sensor=S0,S1,S2,S3,S4 \
   --type=in --proto=17 --sport=123 --pass=NTP-in.raw \
   --print-vol=stdout
     |               Recs|            Packets|                Bytes|      Files|
Total|            3549008|           15506147|           3118288427|         60|
 Pass|               3936|               6148|               493968|           |
 Fail|            3545072|           15499999|           3117794459|           |
```

Example 6.1: Using `rwfilter` to Profile NTP Activity

Differentiate Benign and Malicious Traffic

The second phase of Madison's exploratory analysis examines the UDP activity on port 123 more closely to look for patterns that might differentiate between benign and malicious traffic. `rwuniq` is well suited for this task, since it allows her to look at the traffic over a variety of contingencies. RFC 5905[15] specifies that NTP packets must have a byte size of 76 bytes for a request and 96 bytes for a response. Madison therefore expects to find flow bytes that are a multiple of either of these two values.

Example 6.2 runs `rwuniq` on the output files from Example 6.1. Command 1 shows an excerpt of the various byte counts for outgoing flows. Command 2 shows a similar excerpt for the incoming flows. The byte sizes are the ones that Madison expected: 76, 96, $152 = 2 \times 76$, $192 = 2 \times 96$. (While not shown in the excerpt, all of the other byte sizes are multiples of either 76 or 96.)

What is surprising are the further columns on the last line of the excerpts in Example 6.2. NTP is a protocol where machines on a network make queries to a few servers for time values, then adjust their clocks based

[15] Internet Engineering Task Force (IETF) Request for Comments 5905 (https://www.ietf.org/rfc/rfc5905.txt)

on the answers received. Madison therefore expects outgoing traffic to come from only a few addresses but go to multiple hosts in the local network.

However, the results of her analysis are different from what she expects. For flows with byte size 192 (double the response value), the number of local hosts and the number of remote hosts are nearly equal, which is unusual. Additionally, the number of incoming packets is larger than the number of outgoing requests. This disparity is also unusual. Madison realizes that she needs to further explore this traffic.

```
<1>$ rwuniq --fields=bytes \
   --values=Flows,packets,distinct:dip,distinct:sip --sort \
   NTP-out.raw \
| head -5
     bytes|   Records|         Packets|dIP-Distin|sIP-Distin|
        76|      5728|            5728|         2|        17|
        96|       362|             362|         6|        75|
       152|       563|            1126|         1|         1|
       192|       323|             646|        50|        46|
<2>$ rwuniq --fields=bytes \
   --values=Flows,packets,distinct:dip,distinct:sip --sort \
   NTP-in.raw \
| head -5
     bytes|   Records|         Packets|dIP-Distin|sIP-Distin|
        76|      2083|            2083|        16|         1|
        96|        66|              66|        27|         5|
       152|      1126|            2252|         1|         1|
       192|       615|            1230|        50|        46|
```

Example 6.2: Using `rwuniq` to examine NTP Activity

Plot Anomalous Flows

Now that Madison has identified the anomalous flows, it's time to take a more detailed look at them. Her investigation of these flows is shown in Example 6.3. She first uses `rwfilter` to isolate the incoming 192-byte flows, then calls `rwcount` to generate a time series of that traffic. The results are somewhat lengthy, since the `rwcount` command splits a day of traffic into five-minute bins.

Command 1 uses `rwcount` options to generate a comma-separated-value file (CSV), which she loads into a graphing program (in this case, Microsoft Excel). The resulting time series plot is shown in Figure 6.2.

NTP traffic is normally expected to either cluster around the point at which computers are connecting to the network, or occur periodically thereafter as the computers adjust their clocks. But Figure 6.2 shows that, while there are regular pulses of traffic (visible more to the right end of the timeline), there are also irregular collections of traffic earlier in the day. This is not a normal NTP traffic pattern.

```
<1>$ rwfilter NTP-in.raw --bytes=192-192 --pass=stdout \
| rwcount --bin-size=300 --delim=',' >NTP-in.csv
```

Example 6.3: Using `rwcount` to generate NTP Timelines

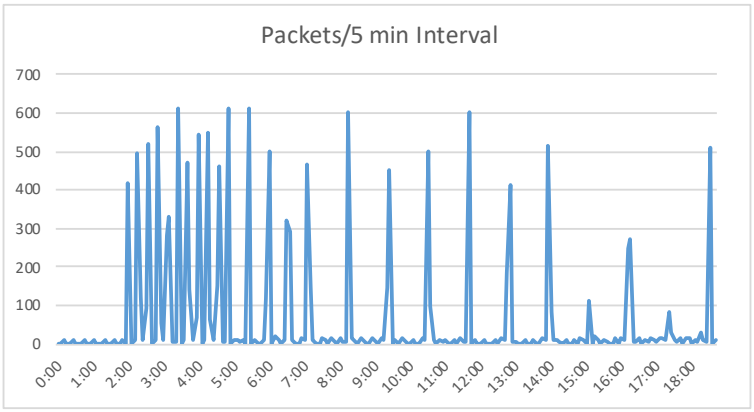

Figure 6.2: Time Series Plot of NTP Traffic

Is the Irregular NTP Traffic Related to Client-side Initialization?

Madison's next step is to determine if this early irregular traffic is related to initialization of the client side of the NTP traffic. Example 6.4 shows how she generates summary bags for the hosts involved.

An obvious approach at this point would be to issue another call to `rwfilter` to pull traffic prior to the start of the NTP traffic for the destinations involved, then call `rwuniq` to summarize the traffic. One complication to this approach is that the start times for the NTP traffic are not aligned. Rather than setting a fixed time (such as the minimum start time), Madison decides that it's wiser to analyze client traffic on a one-by-one basis. This individual approach has two disadvantages: it requires more programming (14 lines of commands, as opposed to two or three), and it is somewhat slower in execution (for the data being examined, about five seconds, rather than an immediate result). However, these disadvantages are outweighed by the increased precision of the results.

Command 1 of Example 6.4 starts this process by identifying the earliest start time for each client (internal recipient) of the NTP traffic, using a combination of `rwfilter` and `rwuniq`. The output of `rwuniq` is formatted for easy parsing, then saved to the file `source-fields.txt`.

Command 2 prepares to analyze traffic for individual clients by creating an empty bag (`cur.bag`). The bag's key is source IP address; its counter is the number of flow records associated with each source IP address.

Command 3 is the loop that reads the start time information for each client (shown by the redirection at the end and the `read` call in the `while` statement's condition). The body of the loop then uses `date` as the NTP start time for the client. It calculates a ten minute time window prior to the start time (using shell arithmetic and `awk` to reformat the range elements as SiLK formatted date-time values).

The bag that was initialized in Command 2 (`cur.bag`) is updated for each source IP address. The loop in Command 3 calls `rwfilter` using the time range for each address to find activity prior to the NTP traffic, using `rwbagtool` to add the results to the bag. Since `cur.bag` is initially empty but keyed by the client IP address, the prior activity is calculated for each client in turn. Although the `rwbagtool` call adds the bags, it is really just inserting the new entry in the summary bag, using `temp.bag` as an intermediate that is renamed in the `mv` call to overwrite the summary bag.

After all of the clients are summarized, Command 4 displays the first 10 entries in `cur.bag`. They represent a brief sample of the 48 lines of full output.

- Low values (in the tens) would support Madison's hypothesis that the host in question is in the process of initialization, as is possible for `192.168.50.11`.
- High values (in the hundreds to thousands) would support her hypothesis that the host in question is in ongoing use, as is the case for most of the hosts.

```
<1>$ rwfilter NTP-in.raw --bytes=192-192 --pass=stdout \
| rwuniq --fields=dip --values=stime-earliest --no-titles \
   --delim=' ' --output=source-fields.txt
<2>$ rwbagbuild --bag-input=/dev/null --key-type=sipv4 \
   --counter-type=records --output=cur.bag
<3>$ while IFS="" read -r line || [[ -n "$line" ]]; \
do srcArray=($line); \
    StEpoch=$( date -d ${srcArray[1]} +"%s"); \
    StTimeE=$(echo $(( $StEpoch - 1 )) \
    | awk '{print strftime("%Y/%m/%dT%T",$1)}'); \
    StTimeS=$(echo $(( $StEpoch - 600 )) \
    | awk '{print strftime("%Y/%m/%dT%T",$1)}'); \
    rwfilter --start=2015/06/16 --sensor=S0,S1,S2,S3,S4 \
    --type=out,outweb --saddress=${srcArray[0]} \
    --stime=${StTimeS}-${StTimeE} --pass=stdout \
    | rwbag --bag-file=sipv4,flows,stdout \
    | rwbagtool --add cur.bag stdin --output=temp.bag; \
    mv temp.bag cur.bag; \
done < source-fields.txt
<4>$ rwbagcat cur.bag | head -10
     10.0.40.54|                1452|
  192.168.40.25|                1777|
  192.168.40.50|                 183|
  192.168.40.51|                 123|
  192.168.50.11|                  63|
192.168.111.109|                 211|
192.168.111.131|                 319|
 192.168.121.57|                 325|
 192.168.121.77|                 291|
192.168.121.145|                 248|
```

Example 6.4: Using `rwuniq` and Bags to Summarize Prior Traffic on NTP Clients

Possible Explanations for Anomalous NTP Traffic

While it is clear that there is a range of flow counts during the ten-minute time window prior to the start of NTP traffic, the numbers are large enough for Madison to eliminate the hypothesis that the anomalous NTP traffic is due to initialization of the clients. The irregular traffic could be due to a variety of causes:

- It could be a covert means of mapping clients: send NTP results to the clients, then see which ones respond with an `ICMP host unreachable` message.
- It could be a covert signaling or command interface for malicious software that is running on the clients.

- It could reflect a flaw in the NTP implementation on the network. However, the relative maturity of NTP and its implementations and the halting of the irregular traffic discount this possibility.
- It could be a reflection attack against the target network. However, the overall level of traffic involved tends to discount that possibility.

Madison can test and evaluate these hypotheses in further rounds of her exploratory analysis!

6.1.4 Observations on Exploratory Analysis

The initial examination of inbound and outbound data in an exploratory analysis is the basis for all that follows. It is the starting point for an accumulating body of experience that aids the analyst in interpreting the results.

The odd behavior shown in Example 6.2 reflects a general aspect of exploratory analysis: it is shaped by the dynamics of network behavior. Since Madison is exploring previously-unexamined behaviors, her process incorporates consideration of new behaviors and trends. This is done by both noting results that are expected in a given situation and those that are surprising.

Surprising results (such as those in Example 6.2) often serve as starting points for further rounds of analysis. Networks are not static: the traffic they carry constantly changes over time. Madison therefore must build an evolving understanding of their behavior.

Madison's exploratory analysis uses a variety of SiLK tools. It also employs tabular data, graphical summary, and calculation of specific values. This diverse set of tools and techniques reflects the open-ended nature of exploratory analysis. She needs to gain different views into the data to get a good understanding of the target of the analysis.

The outcome of an exploratory analysis can take several forms. As the analysis proceeds, it produces more and more background information on the network hosts and their traffic. As this information is captured, it aids in interpreting both results obtained later in the analysis and the results of other analyses.

Background information is not the only thing that can be reused in an exploratory analysis! Its component parts can be applied in other analyses as filters or analytics. For example, the date math portion of Example 6.4 is also incorporated into the case study discussed in Chapter 7. As the body of such filters and analytics expands, further exploratory analyses are easier to perform—simply reuse the ones developed for earlier analyses.

6.2 Exploratory Analysis: Analytics

The SiLK commands described in this chapter involve sophisticated uses of the SiLK tool suite that are often appropriate for exploratory analyses. However, they can be used with any analysis method.

6.2.1 Using Tuple Files for Complex Filtering

Partitioning criteria for many analyses comprise specific combinations of field values, any one of which can be considered as *passing*. While you can make repeated `rwfilter` calls and merge them later, this approach is often inefficient as it may involve pulling the same records from the repository several times.

For example, consider an analysis that is looking for a Simple Network Management Protocol (SNMP) call generated from viewing a malicious email message. SNMP is associated with UDP port 161, email receipt with TCP port 25. The naive approach would be a call to `rwfilter` that simply merges these two ports:

```
rwfilter --protocol=6,17 --dport=25,161
```

This call to `rwfilter` actually uses four permutations of the selection parameters to select records (protocol 6 and dport 25, protocol 6 and dport 161, protocol 17 and dport 25, protocol 17 and dport 161), not just the two that apply for this example (protocol 6 and dport 25, protocol 17 and dport 161).

As shown in Example 6.5, you could use two calls to `rwfilter` to pull the desired port-protocol permutations separately, then combine the data files with the `rwappend` command. However, this approach involves two calls to `rwfilter` that re-read the input flow records. If the analysis includes many cases, the same data would be read many times. On top of that, if the data set is large, each pull from the repository could take a significant amount of time.

```
<1>$ rwfilter --start=2015/06/17 --type=in,out --protocol=6 \
   --dport=25 --pass=result.raw
<2>$ rwfilter --start=2015/06/17 --type=in,out --protocol=17 \
   --dport=161 --pass=part2.raw
<3>$ rwappend result.raw part2.raw
<4>$ rm part2.raw
```

Example 6.5: Using Multiple Data Pulls to Filter on Multiple Criteria

A more efficient approach is to store partitioning criteria as a *tuple file* and use that file with `rwfilter` to pull the records in a single operation. A tuple file is a text file consisting of the five-tuple fields (`sIP`, `dIP`, `sPort`, `dPort`, `protocol`) delimited by vertical bars (`|`). `rwfilter` pulls the flow records that match the entries in the tuple file from the SiLK repository.

To select the protocol-dport combinations of (6,25) and (17,161), you could create a tuple file that contains both combinations:

```
protocol|dport
       6|25
      17|161
```

Running `rwfilter` with the `--tuple-file` switch set to the name of this tuple file will select only those flow records with (protocol 6, dport 25) and (protocol 17, dport 161). Only one call to `rwfilter` would be needed to pull these records, not two (as in Example 6.5).

Using Tuple Files to filter Web Servers

Example 6.6 shows a tuple file that is used to choose web server addresses on different ports.

- Command 1 shows the tuple file, `webservers.tuple`. The first line contains headers that identify the fields associated with the columns, `dIP` and `dPort`. The rest of the tuple file lists the destination IP address and destination port combinations of interest.

- This file can then be used with `rwfilter` as shown in Command 2. The `--tuple-file` option need not be the only partitioning option. In Command 2, the `--protocol` parameter also is specified as a partitioning criterion.

```
<1>$ cat <<EOF >webservers.tuple
             dIP|dPort
     10.0.40.21|  443
     10.0.40.23| 8443
     10.0.20.59|   80
  192.168.20.59|   80
     10.0.40.21|   80
  192.168.40.24|  443
  192.168.40.27|  443
  192.168.40.91|  443
  192.168.40.92|  443
EOF
<2>$ rwfilter --type=in,inweb --start-date=2015/06/02 \
   --end-date=2015/06/18 --dport=80,443,8443 \
   --protocol=6 --flags-all=SAF/SAF,SAR/SAR \
   --tuple-file=webservers.tuple --sensors=S0,S1 \
   --pass=stdout \
| rwuniq --fields=dIP,dPort --value=Records
             dIP|dPort|   Records|
     10.0.20.59|   80|     29720|
     10.0.40.21|  443|    355791|
     10.0.40.23| 8443|     30934|
  192.168.40.91|  443|         6|
  192.168.40.27|  443|         9|
  192.168.40.92|  443|         3|
  192.168.20.59|   80|     10652|
     10.0.40.21|   80|      1565|
  192.168.40.24|  443|        24|
```

Example 6.6: Filtering on Multiple Criteria with a Tuple File

In some cases, you can obtain results more quickly by using seemingly redundant command-line options to duplicate some of the values from the tuple file. For instance, adding `--dport=80,443,8443` to the `rwfilter` call in Example 6.6 reduces the number of records that need to be examined with the tuple file. No matter where they appear in the `rwfilter` call, tuple files are always processed after the parameters that partition based on individual flow record fields, and before those using plug-ins or Python. However, filtering by multiple destination ports is not a substitute for the tuple file in Example 6.6, as these criteria are not sufficiently restrictive to produce the desired results.

6.2.2 Manipulating Bags

Using bags to store key values and counts (as described in Section 4.2.4) can be very helpful to store the intermediate and final results of a multi-path analysis. SiLK supports several advanced options for working with bags. In addition to comparing bags with `rwbagtool` (Section 4.2.4) and extracting sets from bags (Section 4.2.5), you can use `rwbagtool` to do the following:

- add and subtract bags (analogous to the SiLK set operations)
- multiply bags by scalar numbers
- divide bags
- threshold bags (filter bags on volume)

The result of these operations is a bag with new volumes.

Adding and Subtracting Bags

Suppose you want to find the total number of records associated with the IP addresses that are stored in two bags. You can add the contents of the two bags together to create a new bag that holds the sum of their contents.

To add bags together, use the `rwbagtool --add` parameter. The `--output-path` parameter specifies where to deposit the results. Most of the results from `rwbagtool` are bags themselves. Example 6.7 shows how to use bag addition to find the total number of flows of inbound web traffic over a two-day period.

1. The `rwbagcat` calls in commands 1 and 2 display the contents of the two bags to be added, `web-20150616.bag` and `web-20150617.bag`. Each bag contains a day's worth of inbound web traffic flows.
2. Command 3 adds the two bags with the `rwbagtool --add` parameter.
3. The results of the addition are stored in `web-sum.bag`, which is shown in Command 4.
    - If an IP address appears in both bags, the `rwbagtool --add` command sums up the number of flows in the two bags. For instance, the IP address `10.0.20.59` appears in both bags. The number of flows for this IP address in `web-sum.bag` is the sum of the flows in the two bags.
    - If an IP address appears in just one bag, the `rwbagtool --add` command still includes it in the results. For instance, the IP address `190.168.40.27` only appears in the bag `web-20150616.bag` but is included in the results stored in `web-sum.bag`.

Similarly, you may want to subtract the byte counts in one bag from those stored in another bag to find out how many are left over after a step in your analysis. Use the `rwbagtool --subtract` command to subtract the contents of bags.

Bag subtraction operates in the same fashion as bag addition, but all bags after the first are subtracted from the first bag specified in the command. Bags cannot contain negative values: any subtraction resulting in a negative number causes `rwbagtool` to omit the corresponding key from the resulting bag.

```
<1>$ rwbagcat web-20150616.bag
     10.0.20.59|                7977|
     10.0.40.21|              135757|
     10.0.40.23|               11700|
  192.168.20.59|                3980|
  192.168.40.24|                  17|
  192.168.40.27|                   9|
  192.168.40.91|                   3|
<2>$ rwbagcat web-20150617.bag
     10.0.20.59|               15248|
     10.0.40.21|              221599|
     10.0.40.23|               19234|
  192.168.20.59|                6672|
  192.168.40.24|                   7|
  192.168.40.91|                   3|
  192.168.40.92|                   3|
<3>$ rwbagtool --add web-20150616.bag web-20150617.bag \
   >web-sum.bag
<4>$ rwbagcat web-sum.bag
     10.0.20.59|               23225|
     10.0.40.21|              357356|
     10.0.40.23|               30934|
  192.168.20.59|               10652|
  192.168.40.24|                  24|
  192.168.40.27|                   9|
  192.168.40.91|                   6|
  192.168.40.92|                   3|
```

Example 6.7: Merging the Contents of Bags Using `rwbagtool --add`

Hint 6.1: Be Aware of Bag Types with `rwbagtool`

Bags do store information in the file header about which types of keys and counts they contain. However, the information is not used to restrict bag operations. Consequently, `rwbagtool` will add or subtract byte bags and packet bags without warning, producing meaningless results.

If unequal but compatible types are added or subtracted, a meaningful result type will be produced. For example, keys of `sIPv4` and `dIPv4` will produce a result key of type `any-IPv4`. When incompatible types are combined, the resulting type will be `custom` (the most generic bag type). Use `rwfileinfo --fields=bag` to view this information, as described in Section 4.2.9.

Multiplying and Dividing Bags

You can multiply the values in a bag by a scalar number and divide the contents of a bag by the contents of another bag. This is useful for operations such as computing percentages (for instance, to compare traffic levels during different time periods).

- Use the `rwbagtool --scalar-multiply` command to multiply all of the counter values in a bag by a scalar value. Bags can only be multiplied by a scalar value.

- Use the `rwbagtool --divide` command to divide the counter values in one bag by those of another.

Hint 6.2: Use Caution when Dividing Bags with `rwbagtool`

Be very careful when dividing bags. **The second (denominator) bag must contain every key found in the first (numerator) bag—do not divide by zero!** To ensure that the elements of the two bags match, use the `rwbagtool --intersect` command to remove mismatched elements.

1. Extract the set of IP addresses from the denominator bag by using `rwbagtool --coverset` as described in Section 4.2.5.
2. Run `rwbagtool --intersect` on the numerator bag to remove all elements that are not found in the denominator bag as described in Section 4.2.5.
3. Use `rwbagtool --divide` to divide the contents of the numerator bag by those of the denominator bag.

Example 6.8 shows how to use scalar multiplication and division. The example computes the percentage change in traffic between the two bags from Example 6.7. It uses `rwbagtool --coverset` to remove the IP addresses that do not appear in both bags, then uses the `rwbagtool` options `--scalar-multiply` and `--divide` to compute the percentage change in traffic between the IP addresses in both bags.

```
<1>$ rwbagtool --coverset 20150616.bag >20150616.set
<2>$ rwbagtool --coverset 20150617.bag >20150617.set
<3>$ rwsettool --intersect 20150616.set 20150617.set \
   >common.set
<4>$ rwbagtool --scalar-multiply=100 --intersect=common.set \
   20150616.bag >multiply.bag
<5>$ rwbagcat multiply.bag
     10.0.20.59|              797700|
     10.0.40.21|            13575700|
     10.0.40.23|             1170000|
  192.168.20.59|              398000|
  192.168.40.24|                1700|
  192.168.40.91|                 300|
<6>$ rwbagcat 20150617.bag
     10.0.20.59|               15248|
     10.0.40.21|              221599|
     10.0.40.23|               19234|
  192.168.20.59|                6672|
  192.168.40.24|                   7|
  192.168.40.91|                   3|
  192.168.40.92|                   3|
<7>$ rwbagtool --intersect=common.set 20150617.bag \
   >predivide.bag
<8>$ rwbagtool --divide multiply.bag predivide.bag >divide.bag
<9>$ rwbagcat divide.bag
     10.0.20.59|                  52|
     10.0.40.21|                  61|
     10.0.40.23|                  61|
  192.168.20.59|                  60|
  192.168.40.24|                 243|
  192.168.40.91|                 100|
```

Example 6.8: Using `rwbagtool` to Generate Percentages

Thresholding Bags with Count and Key Parameters

Sometimes, you may want to threshold the contents of a bag to items that are larger than a minimum value or smaller than a maximum value. This thresholding can be used to limit key values (e.g., eliminating IP addresses that are lower or higher than the specified address value) as well as count values (e.g., eliminating IP addresses with packet counts that are lower than the desired volume).

The `--minkey`, `--maxkey`, `--mincounter`, and `--maxcounter` parameters supported by `rwbagcat` are also supported by `rwbagtool`. In this case, they specify the minimum and maximum key and count values for output. They can optionally be combined with an operation parameter (e.g., `--add`, `--subtract`) or a masking parameter (i.e., `--intersect` or `--complement-intersect`) to perform other operations on a bag. Example 6.10 shows an example of thresholding by minimum and maximum counts.

6.2.3 Sets Versus Bags: A Scanning Example

Both sets and bags can be employed to search for network scanners. This section provides some examples of each and contrasts how they are used within an analysis.

Fine-tuning IP Sets to Find Scanners

Using IP sets can focus on alternative representations of traffic and identify network scanning and other activities. Example 6.9 drills down on IP sets themselves and provides a different view of this traffic.

```
<1>$ rwfilter --start-date=2015/06/02 --protocol=6 \
   --type=in,inweb --packets=1-3 --pass=stdout \
| rwset --sip-file=low.set
<2>$ rwfilter --start-date=2015/06/02 --protocol=6 \
   --type=in,inweb --packets=4- --pass=stdout \
| rwset --sip-file=high.set
<3>$ rwsettool --difference low.set high.set \
   --output-path=final.set
<4>$ rwsetcat low.set --count-ips
36
<5>$ rwsetcat final.set --count-ips
2
```

Example 6.9: Using `rwset` to Filter for a Set of Scanners

This example isolates the set of hosts that exclusively scan from a group of flow records using `rwfilter` to separate the set of IP addresses that complete legitimate TCP sessions from the set of IP addresses that never complete sessions. As this example shows, the `final.set` set file consists of two IP addresses in contrast to the set of thirty-six that produced low-packet flow records—these addresses are consequently suspicious.[16]

[16] While this might be indicative of scanning activity, the task of scan detection is more complex than shown in Example 6.9. Scanners sometimes complete connections to hosts that respond (to exploit vulnerable machines); non-scanning hosts sometimes consistently fail to complete connections to a given host (contacting a host that no longer offers a service). A more complete set of scan detection heuristics is implemented in the `rwscan` tool, which is discussed in Section 4.2.4.

Using Bags to Find Scanners

To show how bags differ from sets, let's revisit the scanning filter presented in Example 6.9. The difficulty with that code is that if a scanner completed *any* handshake, it would be excluded from the `low.set` file. Many automated scanners would fall under this exclusion if any of their potential victims responded to the scan. It would be more robust to include as scanners hosts that complete only a small number of their connections (10 or fewer) and have a reasonable number of flow records covering incomplete connections (10 or more).

By using bags, Example 6.10 is able to incorporate counts, resulting in the detection of more potential scanners.

1. The calls to `rwfilter` in commands 1 through 3 are piped to `rwbag` to create bags for incomplete, FIN-terminated, and RST-terminated traffic.

2. Commands 4 and 5 use `rwbagtool --coverset` to generate the cover sets for these bags. These commands also use thresholding to generate two sets: `fast-low.set`, which contains only IP addresses with fewer than 10 low-packet connections (`--mincounter=10`) and `fast-high.set`, which contains only IP addresses with more than 10 incomplete connections (`--maxcounter=10`).

3. Command 6 uses `rwsettool --difference` to find the set of IP addresses that are members of `fast-low.set` but not members of `fast-high.set`. The result, `scan.set`, represents the set of IP addresses that responded to the scans.

4. Command 7 uses `rwsetcat` to count the number of IP addresses in each bag.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --type=in,inweb --bytes=2048- --pass=stdout \
| rwfilter stdin --duration=1200- --pass=slowfile.rw \
   --fail=fastfile.rw
<2>$ rwfilter fastfile.rw --protocol=6 --flags-all=S/SRF \
   --packets=1-3 --pass=stdout \
| rwbag --sip-flows=fast-low.bag
<3>$ rwfilter fastfile.rw --protocol=6 \
   --flags-all=SAF/SARF,SR/SRF --pass=stdout \
| rwbag --sip-flows=fast-high.bag
<4>$ rwbagtool fast-low.bag --mincounter=10 --coverset \
   --output-path=fast-low.set
<5>$ rwbagtool fast-high.bag --maxcounter=10 --coverset \
   --output-path=fast-high.set
<6>$ rwsettool --difference fast-low.set fast-high.set \
   --output-path=scan.set
<7>$ rwsetcat fast-low.set fast-high.set scan.set --count-ips
fast-low.set:40
fast-high.set:104
scan.set:37
```

Example 6.10: Using `rwbagtool` to Filter Out a Set of Scanners

6.2.4 Manipulating SiLK Flow Record Files

SiLK provides several commands for manipulating binary flow record files. You can combine these files, split them up into multiple files, and sample them.

Combining Flow Record Files to Provide Context

When you are profiling flow records, you may want to drill down into the data to find specific behaviors. This is especially useful for analyzing traffic with large volumes (for example, by the duration of transfer and by protocol). Issuing repeated `rwfilter` calls subdivides large data sets into smaller ones that are easier to examine and manipulate. However, sometimes this obscures the "big picture" of what is happening during an event. Combining flow record files is one way to provide this kind of context.

Use one of the following SiLK commands to merge multiple flow record files:

- `rwcat` concatenates flow record files in the order in which they are listed. It creates a new flow record file that contains the merged records.
- `rwappend` places the contents of the flow record files at the end of the first specified flow record file. It does not create a new file.

Hint 6.3: SiLK Tools Can Use Multiple Flow Files

As an alternative to combining flow files, many of the SiLK tools accept multiple flow record files as input to a single call. For example, `rwfilter` can accept several flow files to filter during a single call and `rwsort` can accept several flow files to merge and sort during a single call. Very often, it is more convenient to use multiple inputs than to combine flow files.

In Example 6.11, `rwcat` is used to combine previously filtered flow record files to permit the counting of overall values.

1. The initial call to `rwfilter` in Command 1 pulls out all records in the period of interest: 2015/6/10 through 2015/6/24. Subsequent calls to `rwfilter` split these records into three files, depending on the duration of the flow:
   - slow flows of at least 20 minutes duration (i.e., those that match the partitioning criteria of `--duration=1200-`)
   - medium flows of 1–20 minutes duration (i.e., those that match the partitioning criteria of `--duration=60-1199`),
   - fast flows of less than 1 minute duration (the remainder of the flows, which had failed both partitioning criteria and must therefor be less than one minute long)
2. The calls to `rwfilter` in Commands 2 through 4 split each of the initial divisions based on protocol: UDP (17), TCP (6), and ICMPv4 (1). They are saved to files whose names correspond to their speed and protocol (e.g., `slow17.rw`, `med17.rw`, `fast17.rw`).
3. The calls to `rwcat` in commands 5–7 combine the three splits for each protocol into one overall file per protocol (e.g., `all17.rw`).

4. Command 8 is a short script that produces a summary output reflecting the volume of records in each of the composite files.

```
<1>$ rwfilter --type=in,inweb --start-date=2015/6/10 \
   --end-date=2015/6/24 --protocol=0- --note-add='example' \
   --pass=stdout \
| rwfilter stdin --duration=1200- --pass=slowfile.rw \
   --fail=stdout \
| rwfilter stdin --duration=60-1199 --pass=medfile.rw \
   --fail=fastfile.rw
<2>$ rwfilter slowfile.rw --protocol=17 --pass=slow17.rw \
   --fail=stdout \
| rwfilter stdin --protocol=6 --pass=slow6.rw --fail=stdout \
| rwfilter stdin --protocol=1 --pass=slow1.rw
<3>$ rwfilter medfile.rw --protocol=17 --pass=med17.rw \
   --fail=stdout \
| rwfilter stdin --protocol=6 --pass=med6.rw --fail=stdout \
| rwfilter stdin --protocol=1 --pass=med1.rw
<4>$ rwfilter fastfile.rw --protocol=17 --pass=fast17.rw \
   --fail=stdout \
| rwfilter stdin --protocol=6 --pass=fast6.rw --fail=stdout \
| rwfilter stdin --protocol=1 --pass=fast1.rw
<5>$ rwcat slow17.rw med17.rw fast17.rw --output-path=all17.rw
<6>$ rwcat slow1.rw med1.rw fast1.rw --output-path=all1.rw
<7>$ rwcat slow6.rw med6.rw fast6.rw --output-path=all6.rw
<8>$ echo -e "\nProtocol, all, fast, med, slow"; \
for p in 6 17 1; \
   do rm -f ,c.txt ,t.txt ,m.txt
   echo " count-records -" >,c.txt
   for s in all fast med slow; \
     do rwfileinfo $s$p.rw --fields=count-records \
     | tail -n1 >,t.txt
     join ,c.txt ,t.txt >,m.txt
     mv ,m.txt ,c.txt
   done
   sed -e "s/^ *count-records - */$p,/" \
          -e "s/\([^, ]*\),* */\1, /g" ,c.txt
done

Protocol, all, fast, med, slow
6, 6327373, 6280726, 46274, 373,
17, 7304579, 7258750, 45029, 800,
1, 131843, 124221, 6271, 1351,
```

Example 6.11: Combining Flow Record Files with `rwcat` to Count Overall Volumes

Annotation with `rwcat`. When using the `rwfileinfo` command, be aware that `rwcat` creates a new file and can record annotation (using `--note-add` and `--note-file-add`) in the output file header. However, it does not preserve this information from its input files. `rwappend` cannot add annotations and command history to the output file.

Help with `rwcat` and `rwappend`. For more information about the `rwcat` command, type `man rwcat`. For a list of command options, type `rwcat --help`.

For more information about the `rwappend` command, type `man rwappend`. For a list of command options, type `rwappend --help`.

Dividing and Sampling Flow Record Files with `rwsplit`

Some analyses are facilitated by dividing or sampling flow record files. For example, one approach for performing coarse parallelization is to divide a large flow record file into pieces and concurrently analyze each piece separately. For extremely high-volume problems, analyses on a series of robustly taken samples can produce a reasonable estimate using substantially fewer resources. `rwsplit` is a tool that facilitates both of these approaches to analysis.

Each call to `rwsplit` requires the `--basename` switch to specify the base file name for output files. In addition, one of these parameters must be present:

- `--ip-limit` specifies the IP address count at which to begin a new output file
- `--flow-limit` specifies the flow count at which to begin a new output file
- `--packet-limit` specifies the packet count at which to begin a new output file
- `--byte-limit` specifies the byte count at which to begin a new output file

Coarse parallelization with `rwsplit`. Example 6.12 shows a coarsely parallelized process that uses `rwsplit` to divide flow files for processing in parallel.

1. Command 1 pulls a large number of flow records with the `rwfilter` command. It then uses the `rwsplit` command to divide those records into a series of 400,000-record files with a base name of `part`.
2. Command 2 initializes a list of generated filenames. It then uses the `rwfilter` command to separate the records in each file based on sets that contain IP addresses of interest (`mission.set`, `threat.set`, `casual.set`).
3. In Command 3, each of these files is then fed into an `rwfileinfo` call to count the number of records that fall into the selection categories (`mission`, `threat`, and `casual`).

For an additional example of how to use `rwsplit` to improve SiLK performance, see Section 9.6.

Sampling flow files with `rwsplit`. Example 6.13 shows a sampled-flow process. This example estimates the percentage of UDP traffic moving across a large infrastructure over a workday.

1. Command 1 invokes `rwfilter` to perform the initial data pull, retrieving a very large number of flow records. It then uses the `rwsplit` command to pull 100 samples of 1,000 flow records each (`--flow-limit=1000 --sample-ratio=100`), with a 1% rate of sample generation (that is, of 100 samples of 1,000 records, only one sample is retained).

```
<1>$ rwfilter --type=inweb,outweb --start-date=2015/6/10 \
   --end-date=2015/6/24 --bytes-per-packet=45- \
   --pass=stdout \
| rwsplit --flow-limit=400000 --basename=part
# keep track of files generated
<2>$ s_list=(skip); \
for f in part*; \
do n=$(basename $f); \
t=${n%.*}; \
rm -f $t{-miss,-threat,-casual,-other}.rw; \
rwfilter $f --anyset=mission.set --pass=$t-miss.rw \
   --fail=stdout \
| rwfilter stdin --anyset=threat.set --pass=$t-threat.rw \
   --fail=stdout \
| rwfilter stdin --anyset=casual.set --pass=$t-casual.rw; \
s_list=(${s_list[*]} $t{-miss,-threat,-casual}.rw); \
done
<3>$ echo -e "\nPart-name, mission, threat, casual"; \
prev=" "; \
for f in ${s_list[*]}; \
do if [ "$f" = skip ]; \
then continue; \
fi; \
cur=${f%-*}; \
if [ "$prev" != " " ]; \
then if [ "$cur" != "$prev" ]; \
then echo; \
echo -n "$cur, "; \
fi; \
else echo -n "$cur, "; \
fi; \
prev=$cur; \
echo -n $(rwfileinfo --fields=count-records $f | tail -n1 \
| sed -e "s/^ *count-records *//")", "; \
done; \
echo

Part-name, mission, threat, casual
part.00000000, 342291, 2191, 5741,
part.00000001, 337363, 2331, 6036,
part.00000002, 351400, 1365, 3403,
part.00000003, 324637, 1438, 4880,
part.00000004, 59355, 9590, 45748,
part.00000005, 32648, 9042, 43983,
part.00000006, 42775, 13484, 56446,
part.00000007, 52733, 11759, 62119,
part.00000008, 45304, 13089, 60488,
part.00000009, 61153, 3854, 19262,
```

Example 6.12: `rwsplit` for Coarse Parallel Execution

2. Commands 2 through 4 create a file to store the summary results (`udpsample.txt`), then use the `rwstats` command to summarize each sample and isolate the percentage of UDP traffic (protocol 17) in the sample. The results in `udpsample.txt` are then sorted using the operating system `sort` command.

3. Commands 5 through 7 profile the resulting percentages to report the minimum, maximum, and median percentages of UDP traffic.

```
<1>$ rwfilter --type=in,inweb --start-date=2015/6/10 \
   --end-date=2015/6/24 --proto=0-255 --pass=stdout \
| rwsplit --flow-limit=1000 --sample-ratio=100 \
   --basename=sample --max-outputs=100
<2>$ echo -n >udpsample.txt
<3>$ for f in sample*rwf; \
do rwstats $f --values=records --fields=protocol --count=30 \
   --top \
| grep "17|" | cut -f3 "-d|" >>udpsample.txt
done
<4>$ sort -nr udpsample.txt >tmp.txt
<5>$ echo -n "Max UDP%: "; \
   head -n 1 tmp.txt
Max UDP%:  83.700000
<6>$ echo -n "Min UDP%: " ; \
   tail -n 1 tmp.txt
Min UDP%:   1.700000
<7>$ echo -n "Median UDP%: "; \
head -n 50 tmp.txt \
| tail -n 1
Median UDP%:  68.800000
```

Example 6.13: `rwsplit` to Generate Statistics on Flow Record Files

Help with `rwsplit`. For more information about the `rwsplit` command, type `man rwsplit`. For a list of command options, type `rwsplit --help`.

6.2.5 Generate Flow Records From Text

The `rwtuc` (Text Utility Converter) tool creates SiLK flow record files from columnar text. `rwtuc` is effectively the inverse of `rwcut`, with additional parameters to supply values not given by the columnar input.

`rwtuc` is useful when you need to work with tools or scripting languages that manipulate text output. For example, some scripting languages (Perl in particular) have string-processing functions that may be useful during an analysis. However, for later processing, you may need to use a binary file format for compactness and speed. In this situation, you could use `rwcut` to convert the binary flow record files to text, process the resulting file with a scripting language, and then use `rwtuc` to convert the text output back to the binary flow record format.

If scripting can be done in the Python programming language, the programming interface contained in the `silk` module allows direct manipulation of the binary flow records without converting them to text (and

back again). This binary manipulation is more efficient than text-based scripting.[17] See Chapter 8 for more information on using Python with SiLK.

On the other hand, `rwtuc` gives you complete control of the binary representation's content. This is very useful if you need to cleanse a flow record file before exchanging data[18] (for instance, to anonymize IP addresses).

To ensure that unreleasable content is not present in the binary form, an analyst can convert binary flow records to text, perform any required edits on the text file, and then generate a binary representation from the edited text.

Using `rwtuc` to Anonymize Flow Records

Example 6.14 shows a sample use of `rwtuc` for anonymizing flow records. After `rwtuc` is invoked in Command 3, both the file-header information and non-preserved fields have generic or null values.

```
<1>$ rwfilter --sensor=S0 --type=in --start-date=2015/06/02 \
   --end-date=2015/06/18 --protocol=17 \
   --bytes-per-packet=100- --pass=bigflows.rw
<2>$ rwcut bigflows.rw --fields=1-5,stime --num-recs=20 \
| sed -re 's/([0-9]+\.){3}/192.168.200./g' \
   >anonymized.rw.txt
<3>$ rwtuc anonymized.rw.txt --output-path=anonymized.rw
<4>$ rwfileinfo anonymized.rw
anonymized.rw:
  format(id)          FT_RWIPV6ROUTING(0x0c)
  version             16
  byte-order          littleEndian
  compression(id)     lzo1x(2)
  header-length       88
  record-length       88
  record-version      1
  silk-version        3.16.0
  count-records       20
  file-size           512
  command-lines
                  1   rwtuc --output-path=anonymized.rw anonymized.rw.txt
<5>$ rwcut anonymized.rw --fields=sIP,dIP,sTime,sensor \
   --num-recs=4
            sIP|            dIP|                  sTime|sen|
192.168.200.205| 192.168.200.20|2015/06/16T12:50:02.144| S0|
  192.168.200.5| 192.168.200.20|2015/06/16T12:50:03.139| S0|
192.168.200.218| 192.168.200.20|2015/06/16T12:50:05.189| S0|
192.168.200.160| 192.168.200.20|2015/06/16T12:50:09.997| S0|
```

Example 6.14: Simple File Anonymization with `rwtuc`

[17] In several published examples, analysts encoded non-flow information as binary flow records using `rwtuc` or PySiLK so that SiLK commands could be used for the fast filtering and processing of that information.

[18] Ensuring that data content can be shared is quite complex, and involves many organization-specific requirements. `rwtuc` helps with mechanics, but often more transformations are required. The `rwsettool` command contains parameters ending in *-strip* that also help to cleanse IP sets.

Relationship between `rwtuc` and `rwcut`

`rwtuc` expects input in the default format for `rwcut` output. The record fields should be identified either in a heading line or in a `--fields` parameter of the call. `rwtuc` has a `--column-separator` parameter, with an argument that specifies the character-separating columns in the input. For debugging purposes, input lines that `rwtuc` cannot parse can be written to a file or pipe which the `--bad-input-lines` option names.

For fields not specified in the input, an analyst can either let them default to zero (as shown in Example 6.14, especially for `sensor`) or use parameters of the form `--`*`FixedValueParameter`*`=`*`FixedValue`* to set a single fixed value for that field in all records, instead of using zero. Numeric field IDs are supported as arguments to the `--fields` parameter, not as headings in the input file.

Help with `rwtuc`

For more information about the `rwtuc` command, type `man rwtuc`.

For a list of command options, type `rwtuc --help`.

6.2.6 Using Aggregate Bags

Aggregate bags (aggbags for short) are the most complex in a progression of SiLK-related data structures. They are an extension of SiLK bags. Like bags, aggregate bags store key-value pairs. However, bags only store simple key-value pairs, each of which represents a single field or count. Aggregate bags store *composite* key-value pairs: both the key and value can be composed of one or more fields.

For example, the key in an aggregate bag could be a composite of IP address and protocol. The value could be counts of records and bytes for each IP/Protocol combination found in the provided flow records. An aggregate bag stores these combinations of field and count values in a single binary file, as opposed to multiple bag files. Use aggregate bags to summarize and store the results of complex operations in a single file.

Creating Aggregate Bags from Flow Records

To create an aggregate bag that stores the results of a SiLK operation, use the `rwaggbag` command. It creates aggregate bags directly from flow records.

You need to specify the flow record fields that contain the keys and values that are stored in the aggregate bag. For each flow record that `rwaggbag` reads, it extracts the values of the key fields (identified by the `--keys` switch), and combines those fields into a key. It searches for an existing bin that contains the specified key (creating a new bin for that key if none is found), then adds the values for each of the fields listed in the `--counters` switch to the bin's counter. The combined key-value counts are stored in the aggregate bag.

Example 6.15 shows how to use an aggregate bag to count the records associated with source IP addresses that have different protocol and destination port combinations. An ordinary bag can only store one key and one count, limiting it to counting the flow records associated with an IP address, a port, or a protocol. An aggregate bag can count the flow records associated with combinations of these three keys.

1. The `rwfilter` call in Command 1 pulls records with destination ports 1 through 1023.

2. It passes the output to the `rwaggbag` command, which counts the number of flow records associated with the combination of source IP address, destination port, and protocol (`--key=sipv4,dport,protocol --counter=records`).

The new aggregate bag is saved in the file `sim-dport-proto-aggbag`.

Hint 6.4: Sorting and counting aggregate bag keys

The contents of an aggregate bag are sorted and counted in the same order as its keys. For example, the contents of the aggregate bag in Example 6.15 are sorted and counted first by source IP address, then by destination port, and then by protocol.

```
<1>$ rwfilter --start-date=2015/06/17 --type=out --dport=0-1023 \
   --pass=stdout \
| rwaggbag --key=sipv4,dport,protocol --counter=records \
   --output-path=sip-dport-proto.aggbag
<2>$ ls -l sip-dport-proto.aggbag
-rw-r--r--. 1 analyst analyst 6766 Mar 18 14:30 sip-dport-proto.aggbag
<3>$ rwaggbagcat sip-dport-proto.aggbag \
| head -n 5
          sIPv4|dPort|pro|   records|
    10.0.10.254|    0| 89|        48|
     10.0.20.58|   53| 17|   1131799|
     10.0.20.58|  771|  1|       848|
     10.0.20.58|  778|  1|         2|
<4>$ rwfilter --start-date=2015/06/17 --type=out --dport=0-1023 \
   --pass=stdout \
| rwuniq --fields=sip,dport,protocol --sort-output \
| head -n 5
            sIP|dPort|pro|   Records|
    10.0.10.254|    0| 89|        48|
     10.0.20.58|   53| 17|   1131799|
     10.0.20.58|  771|  1|       848|
     10.0.20.58|  778|  1|         2|
```

Example 6.15: Summarizing Source IP, Destination Port, and Protocol with `rwaggbag`

Help with `rwaggbag`. For more information about the `rwaggbag` command, type `man rwaggbag`. For a list of command options, type `rwaggbag --help`.

Viewing Aggregate Bags

Command 3 in Example 6.15 shows how to use the `rwaggbagcat` command to view the contents of a binary aggregate bag as text. The command displays the values of the three keys (`sIPv4`, `dPort`, `pro`) along with the count of records associated with each combination of the three keys.

Help with `rwaggbagcat`. For more information about the `rwaggbagcat` command, type `man rwaggbagcat`. For a list of command options, type `rwaggbagcat --help`.

Comparing `rwaggbag` with `rwuniq`

The SiLK command that most closely matches the capability of `rwaggbag` is `rwuniq`. Command 4 in Example 6.15 compares the output of the `rwuniq` command with that of the `rwaggbag` command. The output from `rwuniq` is the same as the contents of the aggregate bag. The biggest difference is that `rwaggbag` outputs its multi-key counts in binary format; `rwuniq` outputs them as text.

Creating Aggregate Bags from Text

Sometimes, you may not want to use network flow data to create an aggregate bag. For instance, suppose you wish to study incoming ICMP usage on the network. Specifically, you'd like to find out what ICMP types and codes are in use. Are any hosts sending high volumes of ICMP packets?

To facilitate this analysis, you could create a file with the host's source IP, the ICMP type and code, and the count for that triple. An aggregate bag is the best choice, since bags only store pairs of one key and one count and you want to store three keys and a count. However, the man page for the `rwaggbag` command does not show ICMP type and code on the list of key fields.

Instead, use the `rwaggbagbuild` command to store these values and counts. It accepts SiLK key fields as text and produces an aggregate bag file with the desired count of the triple. This gives you more flexibility in creating aggregate bags than the `rwaggbag` command does.

Example 6.16 shows how to create an aggregate bag file that counts and stores the values of this triple.

1. Command 1 creates a text file with the desired fields. The `rwfilter` call pulls network flow data from the desired date range. It pipes the data to the `rwcut` command to output the desired fields as text. We want to include the following values: source IP, ICMP type, and ICMP code (`--fields=sip,itype,icode`). The `rwaggbagbuild` command does not accept a header line with the field names in the file, so we will also use the `--no-titles` option.
2. The `rwaggbagbuild` call in Command 2 takes the text output from `rwcut` and converts it to an aggregate bag. The three keys are specified by `--fields=sIPv4,icmpType,icmpCode`.
3. The `rwaggbagcat` command in Command 3 displays the values of the three keys and the number of flow records counted for each combination.

The IP addresses 10.0.40.20, 4.0.0.34, and 10.0.1.201 are sending high numbers of ICMP messages compared to the other hosts, and should be investigated.

Help with `rwaggbagbuild`. For more information about the `rwaggbagbuild` command, type `man rwaggbagbuild`. For a list of command options, type `rwaggbagbuild --help`.

Thresholding Aggregate Values

The aggregate bag of ICMP type and code counts from Example 6.16 is rather long. We are interested in examining hosts that have produced the most ICMP traffic.

```
<1>$ rwfilter --start-date=2015/06/17 --type=in --protocol=1 \
   --pass=stdout \
| rwcut --fields=sip,itype,icode --no-titles \
   --ipv6-policy=ignore >icmp-no-titles.txt
<2>$ rwaggbagbuild --fields=sIPv4,icmpType,icmpCode \
   --constant-field=record=1 --output-path=icmp.aggbag \
   icmp-no-titles.txt
<3>$ rwaggbagcat icmp.aggbag | head
          sIPv4|icm|icm|   records|
       4.0.0.34|  3|  1|      1459|
     10.0.1.101|  0|  0|       359|
     10.0.1.201|  0|  0|      1002|
      10.0.10.1|  0|  0|       263|
      10.0.10.1| 11|  0|         1|
     10.0.20.58|  3| 10|        19|
      10.0.30.1| 11|  0|         1|
     10.0.40.20|  0|  0|      9440|
     10.0.40.27|  3| 10|         6|
```

Example 6.16: Summarizing Source IP, ICMP Type, and ICMP Code with `rwaggbagbuild`

The `rwaggbagtool` command lets you manipulate aggregate bags, similar to how the `rwbagtool` command manipulates bags. Use the thresholding feature of `rwaggbagtool` to threshold aggregate bag entries by count.

`--min-field=`*`FIELD=VALUE`* selects entries where the specified field is above a minimum value.

`--max-field=`*`FIELD=VALUE`* selects entries where the specified field is below a maximum value.

In this case, we will specify that the value of the records field be greater than or equal to 1000—as shown in Example 6.17.

1. The call to `rwaggbagtool` sets a threshold of 1000 records (`--min-field=records=1000`) and creates a new aggregate bag, `top-icmp.aggbag`.
2. The call to `rwaggbagcat` displays the thresholded aggregate bag.

Help with `rwaggbagtool`. For more information about the `rwaggbagtool` command, type `man rwaggbagtool` or enter the command `rwaggbagtool --help`.

```
<4>$ rwaggbagtool --min-field=records=1000 \
   --output-path=top-icmp.aggbag icmp.aggbag
<5>$ rwaggbagcat top-icmp.aggbag | head
          sIPv4|icm|icm|   records|
       4.0.0.34|  3|  1|      1459|
     10.0.1.201|  0|  0|      1002|
     10.0.40.20|  0|  0|      9440|
     10.0.40.27|  8|  0|      2214|
  192.168.40.20|  0|  0|      1486|
  192.168.40.27|  8|  0|     17354|
  192.168.70.10|  3|  3|      2797|
192.168.110.254|  3|  1|      2856|
192.168.120.254|  3|  1|      2290|
```

Example 6.17: Thresholding an aggregate bag with `rwaggbagtool`

Exporting Aggregate Bags to Bags and IPsets

You can also use the `rwaggbagtool` to export bags and IPsets from aggregate bags. The previous example created an aggregate bag that contains the source IP, ICMP type, and ICMP code triples with the highest record counts. Exporting it allows you to create a bag that holds only one key-value pair—for instance, if you only want to look at high ICMP traffic by source IP and not the other two keys.

Example 6.18 shows how to use the `rwaggbagtool` command to export a bag from an aggregate bag.

1. The call to `rwaggbagtool` selects the key and count to be exported—in this case, source IP and record count (`--to-bag=sIPv4,records`) It saves the exported key-value pairs to a bag file, `top-icmp.bag`.
2. The call to `rwbagcat` displays this file as text.

```
<6>$ rwaggbagtool --to-bag=sIPv4,records \
   --output-path=top-icmp.bag top-icmp.aggbag
<7>$ rwbagcat top-icmp.bag | head
       4.0.0.34|                1459|
     10.0.1.201|                1002|
     10.0.40.20|                9440|
     10.0.40.27|                2214|
  192.168.40.20|                1486|
  192.168.40.27|               17354|
  192.168.70.10|                2797|
192.168.110.254|                2856|
192.168.120.254|                2290|
192.168.130.254|                2032|
```

Example 6.18: Extracting a bag from an aggregate bag with `rwaggbagtool`

Similarly, Example 6.19 shows how to use the `rwaggbagtool` command to export an IPset from an aggregate bag. Creating an IPset of hosts that send the highest amount of ICMP messages can be useful for further analysis of their behavior.

1. The call to `rwaggbagtool` selects the IP addresses to be exported (`--to-ipset=sIPv4`) to a new IPset file, `top-icmp.set`.

2. The `rwsetcat` command displays the contents of this set as text.

```
<8>$ rwaggbagtool --to-ipset=sIPv4 --output-path=top-icmp.set \
   top-icmp.aggbag
<9>$ rwsetcat top-icmp.set | head
4.0.0.34
10.0.1.201
10.0.40.20
10.0.40.27
192.168.40.20
192.168.40.27
192.168.70.10
192.168.110.254
192.168.120.254
192.168.130.254
```

Example 6.19: Extracting an IPset from an aggregate bag with `rwaggbagtool`

6.2.7 Labeling Data with Prefix Maps

Some analyses are easier to conceptualize and perform when you assign a text label to data of interest. You can identify this data and think of it in context by the label you've given it, not just as an abstract group of flow records. You can then filter and perform other operations on the data according to how it is labeled.

SiLK allows you to create a *prefix map* (often abbreviated as *pmap*) to assign user-defined text labels to ranges of IP addresses or protocols and ports. You can use the resulting pmap file to retrieve, partition, sort, count, and perform other operations on network flow records by their labels. This enables you to work with data semantically.

What Are Prefix Maps?

A prefix map file is a binary file that maps a value (either an IP address or a protocol-port pair) to a text label. SiLK supports two general types of prefix maps.

- **User-defined prefix maps** assign arbitrary text labels to IP addresses or protocol-port combinations.
- **Predefined prefix maps** are specialized prefix maps that are typically included with the SiLK distribution and facilitate analysis by country code or traffic direction.

Both types of pmap can be used interchangeably with the `rwfilter`, `rwcut`, `rwsort`, `rwuniq`, `rwstats`, `rwgroup`, and `rwpmaplookup` commands to perform operations on SiLK network data according to how it is labeled.

Comparing Prefix Maps to Sets, Bags, and Aggregate Bags. Like SiLK IP sets, bags, and aggregate bags, prefix maps allow you to create user-defined groups of IP addresses to facilitate further analysis. However, prefix maps are more generalized groupings than sets, bags, and aggregate bags. A set creates a binary association between an IP address and a condition (an address is either in the set or not in the set), a bag between an IP address and a single numeric value, and an aggregate bag between an IP address and a composite (or aggregate) value. However, a prefix map assigns arbitrary, user-defined text labels to many different address ranges. Prefix maps also expand the type of groupings to include assigning labels to protocol-port ranges. It is an entirely different way to identify and group data.

It is often easier to examine IP addresses by label rather than by whether they belong to a bag or set. For example, suppose you want to look at traffic from IP addresses that are linked to different types of malware. You could create multiple IP sets or bags that contain the addresses associated with each type of malware and compute traffic statistics individually for each set or bag. However, this would be cumbersome and time-consuming to script.

Alternatively, you could create a single, user-defined pmap file that labels the addresses by the type of malware that is associated with each address. You could then use the pmap file to filter network flow data and compute traffic statistics according to the type of malware, all in a single analytic. This is a more intuitive and easier way to perform that type of analysis.

Creating a User-defined Prefix Map From a Text File

To create a binary prefix map from a text file, use the `rwpmapbuild` tool. Creating a user-defined pmap is a two-step process:

1. Create a text file that contains the mapping of IP addresses or protocol-port pairs to their labels.
2. Use the `rwpmapbuild` command to translate this text-based file into a binary pmap file.

Creating the Text File. Each mapping line in the text file contains either an IP address or protocol-port pair range with a corresponding label. They are separated by whitespace (spaces and tabs). Include the following information when creating a text file for use with the `rwpmapbuild` command:

- The input file may specify a name for the pmap via the line `map-name` *mapname* where *mapname* is the name of the pmap. Pmap names cannot contain whitespaces, commas, or colons.
- `mode` specifies the type of pmap.
  - `ipv4` creates a pmap containing IPv4 addresses
  - `ipv6` creates a pmap containing IPv6 addresses
  - `proto-port` creates a pmap containing protocol-port pairs
- If you are creating an IP address pmap, specify an address range with either a CIDR block or a whitespace-separated low IP address and high IP address (formatted in canonical form or as integers). Specify a single host as a CIDR block with a prefix length of 32 for IPv4 or 128 for IPv6.
- If you are creating a protocol-port pmap, specify a range that is either a single protocol or a protocol and a port separated by a slash character (`/`). If the range is a single port, specify that port number as the starting and ending value of the range. For example, `17/17` specifies general UDP traffic; `17/53`

`17/53` specifies DNS traffic (UDP on port 53), and `17/67 17/68` specifies DHCP client and server traffic (UDP on ports 67 and 68). A detailed example of a protocol-port pmap is shown in Section 7.3.1.

- Do not use commas, which invalidate the pmap for use with `rwfilter`.
- Comment lines begin with the pound or hashtag character (`#`). Do not use this character in text labels.
- The input file may also contain a *default label* to be used when there is no matching range in the text file. This default is specified by the line `default` *deflabel*, where *deflabel* is the text label specified by the analyst for otherwise-unlabeled address ranges.

For more information about the `rwpmapbuild` command, type `man rwpmapbuild` or enter the command `rwpmapbuild --help`.

Building the Prefix Map File. Example 6.20 shows an example of how to create an IPv4 prefix map of FCC network labels from the FCCX dataset described in Section 1.8. The FCC network description is contained in the file `fccnets.pmap.txt`. It associates network address ranges with text labels that identify their subnetwork locations (`Div0Ext`, `Div1Ext`, etc.).

In addition to the address list, the text file specifies the prefix map name (`map-name fccnets`). This name is used in SiLK commands to identify which prefix map is being used. Using the map name as the name of the text file and resulting pmap file helps you to keep track of and organize your user-defined prefix maps. Note also that the text file includes a default label (`default None`) that is assigned to IP addresses that are not listed in the FCC network description.

The `rwpmapbuild` command takes `fccnets.pmap.txt` as input to create a binary pmap file, `fccnets.pmap`.

```
<1>$ cat <<-EOF >fccnets.pmap.txt
map-name fccnets
default None

# FCC network descriptions
10.0.10.0/24 Div0Ext
10.0.20.0/24 Div0Ext
10.0.30.0/24 Div0Ext
10.0.40.0/24 Div0Ext
10.0.50.0/24 Div0Ext
192.168.10.0/24 Div1Ext
192.168.20.0/24 Div1Ext
192.168.30.0/24 Div1Ext
192.168.40.0/24 Div1Ext
192.168.50.0/24 Div1Ext
192.168.60.0/24 Div1Ext
192.168.70.0/24 Div1Ext
192.168.110.0/23 Div1Ext
192.168.120.0/23 Div1Ext
192.168.122.0/23 Div1Ext
192.168.124.0/24 Div1Ext
192.168.130.0/23 Div1Ext
192.168.140.0/23 Div1Ext
192.168.142.0/23 Div1Ext
192.168.150.0/23 Div1Ext
```

```
192.168.160.0/23 Div1Ext
192.168.162.0/23 Div1Ext
192.168.164.0/23 Div1Ext
192.168.166.0/24 Div1Ext
192.168.170.0/24 Div1Ext
10.0.40.0/24 Div0Int
10.0.50.0/24 Div0Int
192.168.20.0/24 Div1Int1
192.168.40.0/24 Div1Int1
192.168.50.0/24 Div1Int1
192.168.60.0/24 Div1Int1
192.168.70.0/24 Div1Int1
192.168.110.0/23 Div1Int1
192.168.120.0/23 Div1Int1
192.168.122.0/23 Div1Int1
192.168.124.0/24 Div1Int1
192.168.130.0/23 Div1Int1
192.168.140.0/23 Div1Int1
192.168.142.0/23 Div1Int1
192.168.150.0/23 Div1Int1
192.168.160.0/23 Div1Int1
192.168.162.0/23 Div1Int1
192.168.164.0/23 Div1Int1
192.168.166.0/24 Div1Int1
192.168.170.0/24 Div1Int1
192.168.60.0/24 Div1Int2
192.168.110.0/23 Div1Int2
192.168.120.0/23 Div1Int2
192.168.122.0/23 Div1Int2
192.168.124.0/24 Div1Int2
192.168.130.0/23 Div1Int2
192.168.140.0/23 Div1Int2
192.168.142.0/23 Div1Int2
192.168.150.0/23 Div1Int2
192.168.160.0/23 Div1Int2
192.168.162.0/23 Div1Int2
192.168.164.0/23 Div1Int2
192.168.166.0/24 Div1Int2
192.168.170.0/24 Div1Int2
192.168.121.0/24 Div1log1
192.168.122.0/24 Div1log2
192.168.123.0/24 Div1log3
192.168.124.0/24 Div1log4
192.168.141.0/24 Div1ops1
192.168.142.0/24 Div1Ext0
192.168.143.0/24 Div1Ext1
192.168.40.0/24 Div1Ext2
192.168.111.0/24 Div1Ext3
192.168.20.0/24 Div1Ext4
192.168.164.0/24 Div1Ext5
192.168.166.0/24 Div1Ext6
192.168.165.0/24 Div1Ext7
192.168.50.0/24 Div1Ext8
192.168.161.0/24 Div1Ext9
```

```
192.168.162.0/24 Div0Int0
192.168.163.0/24 Div0Int1
EOF
<2>$ rwpmapbuild --input-file=fccnets.pmap.txt \
   --output-file=fccnets.pmap
<3>$ file fccnets.pmap.txt fccnets.pmap
fccnets.pmap.txt: ASCII text
fccnets.pmap:     SiLK, PREFIXMAP v2, Little Endian, Uncompressed
```

Example 6.20: Using `rwpmapbuild` to Create a FCC Pmap File

Help with `rwpmapbuild`. For a list of command options, type `rwpmapbuild --help` at the command line. For more information about the `rwpmapbuild` command, type `man rwpmapbuild`.

Predefined Prefix Maps: Country Codes and Address Types

SiLK has two predefined prefix maps to facilitate common analyses: filtering by country codes and by address types (internal, external, non-routable). They can be used as input to SiLK commands just like user-defined pmaps.

Filtering by Country Code. Country codes identify the nations where IP addresses are registered and are used by the Root Zone Database (e.g., see https://www.iana.org/domains/root/db). They are described in a `country_codes.pmap` file in the `share` directory underneath the directory where SiLK was installed or in the file specified by the `SILK_COUNTRY_CODES` environment variable.

If the current SiLK installation does not have this file, either contact the administrator that installed SiLK or look up this information on the Internet.[19] Country code maps are not in the normal pmap binary format and cannot be built using `rwpmapbuild`.

Filtering Internal, External, and Non-routable Addresses. For common separation of addresses into specific types, normally internal versus external, a special pmap file may be built in the `share` directory underneath the directory where SiLK was installed. This file, `address_types.pmap`, contains a list of CIDR blocks that are labeled `internal`, `external`, or `non-routable`.

The `rwfilter` parameters `--stype` or `--dtype` use this pmap to isolate internal and external IP addresses. The `rwcut` parameter `--fields` specifies the display of this information when its argument list includes `sType` or `dType`. A value of 0 indicates `non-routable`, 1 is `internal`, and 2 is `external`. The default value is `external`.

Using Prefix Maps to Filter Flow Records with `rwfilter`

You can use prefix maps to filter network flow records with the `rwfilter` command. This allows you to pull and partition SiLK repository data based on how it is labeled.

The pmap provides context for filtering network traffic and allows you to perform complex filtering operations in a single analytic. For example, you could create a user-defined pmap that labels IP addresses in network

[19] One such source is the GeoIP® Country database or free GeoLite™ database created by MaxMind® and available at https://www.maxmind.com/; the SiLK tool `rwgeoip2ccmap` converts this file to a country-code pmap.

spaces of interest (such as the one created in Example 6.20) and filter records based on their subnetworks. You could use the predefined country code pmap (`country_codes.pmap`) to filter source or destination IP addresses based on their country of origin. You could create a user-defined port-protocol pmap to filter records for specific port and protocol combinations in order to search for unusual protocols. These types of analysis are easier to perform with data that is labeled via prefix map.

`rwfilter` supports four pmap parameters.

- `--pmap-file` specifies which compiled prefix map file to use and optionally associates a mapname with that pmap. **This switch must be specified before the other prefix map switches.**
- `--pmap-any-`*mapname* specifies the the set of labels used to filter records based on both source and destination IP addresses. Any source or destination IP addresses that matches an IP address with the specified label passes the filter.
- `--pmap-src-`*mapname* and `--pmap-dst-`*mapname* specify the set of labels for filtering by source or destination IP address, respectively. *mapname* is the name given to the pmap during construction or in `--pmap-file`. The `--pmap-file` parameter must come before any use of the *mapname* in other parameters.

Example 6.21 shows how to use the `fccnets.pmap` file from Example 6.20 to select flow records associated with web traffic from hosts on the subnetwork `Div1Int1` in the FCC network.

1. The initial call to `rwfilter` pulls all flow records from the date of interest (`--start-date=2015/06/17`) that contain inbound and outbound web traffic (`--type=inweb,outweb`).
2. The output goes to a second `rwfilter` command that uses the `--pmap-file` parameter to load the file `fccnets.pmap` and create the mapname `fccnets`. The `--pmap-any-fccnets=Div1Int1` parameter filters for all flow records whose source IP address or destination IP address matches any of the IP addresses with the label `Div1Int1` in the pmap file. These source and destination IP addresses represent traffic that flows into, within, and out of the hosts on the `Div1Int1` subnet. (Note that the `--pmap-file` parameter comes before the `--pmap-any` parameter.)
3. Records that pass the pmap-based filter are saved in `fccnets.rw`.

```
<1>$ rwfilter --start-date=2015/06/17 --type=inweb,outweb \
   --protocol=6 --pass=stdout \
| rwfilter stdin --pmap-file=fccnets:fccnets.pmap \
   --pmap-any-fccnets=Div1Int1 --pass=fccnets.rw
<2>$ rwfileinfo fccnets.rw --fields=count-records
fccnets.rw:
  count-records       722193
```

Example 6.21: Using Pmap Parameters with `rwfilter`

Displaying Prefix Labels with `rwcut`

To view the actual value of a prefix map label, use the `rwcut` command with the `--pmap-file` parameter. It takes an argument of a filename or a map name coupled to a filename with a colon. The map name typically comes from the argument for `--pmap-file`; if none is specified there, the name in the source file applies.

The `--pmap-file` parameter adds `src-`*`mapname`* and `dst-`*`mapname`* as arguments to `--fields`. Essentially, it tells `rwcut` to treat the pmap value as just another flow record field. The `--pmap-file` parameter must precede the `--fields` parameter. The two pmap fields display labels associated with the source and destination IP addresses.

Example 6.22 shows how to display the prefix labels for IP addresses in the flow record file `fccnets.rw` (created in Example 6.21). The names of the pmap and its file (`--pmap-file=fccnets:fccnets.pmap`) are specified.

```
<1>$ rwcut fccnets.rw --pmap-file=fccnets:fccnets.pmap \
   --fields=src-fccnets,sPort,dIP,dPort --num-recs=5
src-fccnets|sPort|            dIP|dPort|
   Div1Int1|50373|     10.0.20.59|   80|
   Div1Int1|63440|     10.0.20.59|   80|
   Div1Int1|57862|     10.0.20.59|   80|
   Div1Int1|62669|     10.0.20.59|   80|
   Div1Int1|54211|     10.0.20.59|   80|
```

Example 6.22: Viewing Prefix Map Labels with `rwcut`

Sorting, Counting, and Grouping Records with Prefix Maps

You can also count, sort, and group flow records by prefix label. This lets you work with network data according to how it is labeled in the prefix map, which can be easier and more intuitive than some of the other methods for summarizing data. The `rwsort`, `rwgroup`, `rwstats`, and `rwuniq` tools all work with prefix maps. The prefix map parameters are the same as those used in the `rwcut` command (see Section 6.2.7) and are used to and sort, group, and count records according to the labels in the prefix map file.

Sorting records by prefix label with `rwsort`. Example 6.23 sorts flow records by the prefix value defined in `fccnets.pmap` for source IP addresses and bytes (`--fields=src-fccnets,bytes`). It then uses the `rwcut` command to display the pmap labels and port numbers. The first five records in the data file are displayed.

Notice that the results in Example 6.23 list `None` as their label. This is the default label from Example 6.20 that was assigned to IP addresses that are not included in the pmap. It indicates that these source IP addresses do not belong to the FCC network space. However, they exchanged traffic with hosts that are labeled as belonging to `Div1Int1` on the FCC network. Because the `rwfilter` command in Example 6.21 filtered for *all* records that contained IP addresses labeled as `Div1Int1` in the pmap, it included records where either the source or destination IP address was not part of this subnetwork.

Counting records by prefix label with `rwuniq`. Example 6.24 shows how to count the number of records in the file `fccnets.rw` with labels defined in `fccnets.pmap`. Records without a label are listed as `None`. It also displays the destination port and the number of distinct destination IP addresses.

Again, the `rwuniq` command counts records with source or destination IP addresses that are not part of the `Div1Int1` subnet. Because these hosts communicated with hosts on the `Div1Int1` subnet, they are included in the count. The records in this case all are filtered to be TCP traffic, which makes the port numbers meaningful. TCP and UDP are the main protocols that use ports, and so are the most common ones for

```
<1>$ rwsort fccnets.rw --pmap-file=fccnets:fccnets.pmap \
   --fields=src-fccnets,bytes \
| rwcut --pmap-file=fccnets.pmap --fields=src-fccnets,sport \
   --num-recs=5
src-fccnets|sPort|
       None|55862|
       None|55862|
       None|54392|
       None|54393|
       None|54394|
```

Example 6.23: Sorting by Prefix Map Labels

port-protocol pmaps. SiLK does encode ICMP type and Code into the destination port (sometimes the source port), so ICMP port-protocol pmaps are also used by some analysts.

```
<1>$ rwuniq fccnets.rw --pmap-file=fccnets:fccnets.pmap \
   --fields=src-fccnets,dPort --values=flows,dIP-Distinct \
| head -n 5
src-fccnets|dPort|   Records|dIP-Distin|
    Div0Ext|52963|         8|         1|
       None|57113|         4|         1|
       None|58924|        12|         1|
   Div1Int1|55777|         2|         1|
```

Example 6.24: Counting Records by Prefix Map Labels

Querying Prefix Map Labels

When using prefix maps, you may need to look up which labels correspond to specific IP addresses or protocol-port pairs. Use the `rwpmaplookup` command to query prefix map files—either user-defined pmaps or one of the two predefined pmaps that often are created as part of the SiLK installation (country codes and address types).

You can query a pmap with `rwpmaplookup` by doing one of the following:

- specify the addresses or protocol-port pairs in a text file (the default),
- use the `--ipset-files` parameter to query the addresses in one or more IP sets,
- use the `--no-files` parameter to list the addresses or protocol-port pairs to be queried directly on the command line.

In any of these cases, one and only one of `--country-codes`, `--address-types`, or `--map-name` is used.

IP addresses are specified as described earlier in this section. For protocol-port pmaps, only the names of text files having lines in the format *protocolNumber*`/`*portNumber* or the `--no-files` parameter followed by strings in the same format are accepted. *protocolNumber* must be an integer in the range 0–255, and *portNumber* must be an integer in the range 0–65,535.

If the prefix map being queried is a protocol-port pmap, it makes no sense to query it with an IP set. `rwpmaplookup` prints an error and exits if `--ipset-files` is given.

Example 6.25 shows how to use `rwpmaplookup`.

- Command 1 creates a list of IP addresses and stores it in the file `ips_to_find`.
- Command 2 uses `rwpmaplookup` to find the country codes associated with these addresses. If an IP address is not listed in the country code pmap, the command returns `--` as its value.
- Command 3 looks up the address types of the IP addresses. The first address has a value of 1, indicating an `internal` address. The second address has a value of 2, indicating an `external` address. The third address has a value of 0, indicating a `non-routable` address.
- Commands 4 and 5 build a protocol-port prefix map.
- Command 6 looks up protocol-port pairs from the command line.

```
<1>$ cat <<END_FILE >ips_to_find
192.88.209.244
128.2.10.163
127.0.0.1
END_FILE
<2>$ rwpmaplookup --country-codes ips_to_find
            key|value|
 192.88.209.244|   us|
   128.2.10.163|   us|
      127.0.0.1|   --|
<3>$ rwpmaplookup --address-types ips_to_find
            key|value|
 192.88.209.244|    1|
   128.2.10.163|    2|
      127.0.0.1|    0|
<4>$ cat <<END_FILE >mini_icmp.pmap.txt
map-name miniicmp
default Unassigned
mode proto-port
1/0    1/0      Echo Response
1/768  1/768    Net Unreachable
1/769  1/769    Host Unreachable
1/2048 1/2048   Echo Request
END_FILE
<5>$ rwpmapbuild --input-file=mini_icmp.pmap.txt --output-file=mini_icmp.pmap
<6>$ rwpmaplookup --map-file=mini_icmp.pmap --no-files 1/769 1/1027
      key|           value|
    1/769|Host Unreachable|
   1/1027|      Unassigned|
```

Example 6.25: Query Addresses and Protocol/Ports with `rwpmaplookup`

Help with `rwpmaplookup`. For more information about the `rwpmaplookup` command, type `man rwpmaplookup` or enter the command `rwpmaplookup --help`.

6.3 Exploratory Analysis for Situational Awareness

Exploratory analyses provide the most flexibility in supporting situational awareness. These analyses, with their iterative nature, support differential and actionable awareness (described in Section 2.3.1). While multi-path analyses can identify differences between "what should be" and "what is" (differential awareness) exploratory analyses dig deeper into these differences. An exploratory analysis for situational awareness is well-suited to developing actionable awareness: determining the action to take (if needed) in response to these differences.

6.3.1 Structuring an Exploratory Analysis for Situational Awareness

Exploratory analyses often employ a looser structure than single- or multi-path analyses. For situational awareness, an exploratory analysis isolates and follows differences in characteristics that support a decision about how to maintain and improve network security. The behavior being explored often drives the structure of the analysis. Analysts may work both interactively and progressively to focus, characterize, and draw conclusions from their exploration of network traffic. This iterative process continues until a sufficient level of activity is identified, further data produces no new results, or the time for analysis is exhausted.

Exploratory analysis methods that are useful for investigating network situational awareness are listed below. This is not an exhaustive list of analysis techniques. Once the analyst works through the analysis process interactively, constructing a scripted version of these methods is fairly straightforward.

These exploratory analysis techniques are not exclusive to situational awareness. They can be applied to a wide variety of analyses.

Manifolds

Manifolds chain together calls to `rwfilter` to progressively categorize traffic. An exploratory situational awareness analysis might use a manifold to divide traffic into a series of cases via `rwfilter` commands that link either through `--pass` or `--fail` (exclusive cases) or through `--all` (overlapping cases). These cases can be saved to a file or fed into a tool such as `rwcount`, `rwstats`, or `rwuniq` to generate parallel summaries. The last exclusive case, or "fall-through," is often saved to a file for further exploration.

In an exploratory analysis for situational awareness, the identified cases provide insight into continuance of expected network behavior. The fall-through case often provides insight into unexpected behavior or noise.

For more information on how to set up and use manifolds as part of an analysis, see Section 4.2.1.

Progressive Exclusion

Progressive exclusion involves iteratively removing cases from the analysis to focus on the behavior that is actionable. This approach focuses on isolating traffic that is different enough to support decisions, then generating counts or statistics involving that traffic to clarify a preferred option.

To exclude different types of traffic, use `rwfilter` with a variety of parameters. The specific parameters depend on the nature of the behavior that is being explored. Examples of progressive exclusion with `rwfilter` include the following:

- whitelisting known-good addresses via `--sipset` or `--dipset`
- removing scanning flows via `--flags-all=S/SRF` or `--bytes-per=0-60`,
- eliminating single-packet flows via `--packets=1-1`

Progressive Broadening

Progressive broadening involves iteratively modifying `rwfilter` parameters to pull more data and include more cases. The goal is to capture the desired behaviors more completely. This approach often starts with an initial observation of unusual or suspicious behavior, which is then expanded to provide more data to support a decision. Broadening continues until either no new relevant cases are included, or too many irrelevant cases are included.

For example, progressive broadening can capture patterns where single-packet flows shift to low-packet flows. An initial data pull with `rwfilter --packets=1-1` captures single-packet flows. It can be followed by a call to `rwfilter --packets=1-5` to broaden the analysis to capture low-packet flows.

Analytical Chaining

Analytical chaining involves iteratively pivoting an investigation of flow characteristics to explore different aspects of a suspect behavior. By pivoting from one network characteristic to another, the analyst can follow leads to isolate unusual or suspicious behavior and contrast it against related network activity. Often, this approach starts with a count that is irregular, and then moves through a variety of endpoints and characteristics to provide insight into a behavior of interest.

An example of analytical chaining would be to first examine patterns of behavior associated with a set of addresses via `--saddress` or `--daddress`. The analyst would then switch to examining the same dataset to find behavior associated with ports via `--sport` or `--dport`. An extended example of analytical chaining is presented in Section 6.3.3.

6.3.2 Situational Awareness for User-driven and Automated Services

Most current threats seem to focus on user-driven services (such as web browsing, email, or file handling). However, some still focus on automated processes within the network stack or operating system. Analysts can apply an exploratory analysis workflow to gain better situational awareness of threats to both user-driven and automated services.

For some services, the differences in behavior are fairly obvious. In Example 6.26, the `rwfilter` call in Command 1 pulls web flows (`--type=in,inweb,out,outweb`), which are almost entirely driven by human users. While the three-hour summaries produced by the `rwcount` command do not show much detail, the results do show a twelve-hour usage span, with the central three hours (primary work time) the most active. The `rwfilter` call in Command 2 pulls NTP flows (`--proto=6,17` and `--dport=123`), which are almost entirely driven by the operating system. In this case, the three hour summaries occur throughout the 24-hour day. The results show some variation, but not to the extent of the web flows in Command 1.

```
<1>$ rwfilter --start=2015/06/17 --end=2015/06/17 \
   --type=in,inweb,out,outweb --application=80,443 \
   --pass=stdout \
| rwcount --bin-size=10800
                Date|         Records|                Bytes|           Packets|
2015/06/17T12:00:00|        21300.68|         405172522.46|         644181.85|
2015/06/17T15:00:00|       177508.99|        3846261415.01|        5851526.65|
2015/06/17T18:00:00|        45434.33|         894026650.53|        1409842.50|
<2>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=in,out \
   --proto=6,17 --dport=123 --pass=stdout \
| rwcount --bin-size=10800
                Date|         Records|                Bytes|           Packets|
2015/06/17T00:00:00|         5727.00|            767056.00|           8636.00|
2015/06/17T03:00:00|         9001.00|           1220704.00|          13349.00|
2015/06/17T06:00:00|         5632.00|            721108.00|           8143.00|
2015/06/17T09:00:00|         4914.00|            609168.00|           6978.00|
2015/06/17T12:00:00|         4198.89|            509871.31|           5991.46|
2015/06/17T15:00:00|         3974.83|            494391.91|           5893.05|
2015/06/17T18:00:00|         2779.27|            298084.78|           3538.48|
2015/06/17T21:00:00|         1630.00|            128052.00|           1672.00|
```

Example 6.26: Contrasting User-driven vs. Autonomic Flows with `rwcount`

For other services, the differences are more subtle—particularly when attackers attempt exploits. To separate traffic involving user-driven and automated processes, pull multiple pools of traffic using a variety of criteria:

- port and protocol or application labeling
- initially pulling date-time groups, then later pulling packet sizes and ports
- pulling data according to the endpoints involved

By profiling these datasets and comparing the results for user-driven and automatic services, analysts can identify specific differences that form the basis for further iterations of exploration.

6.3.3 Exploratory Analysis for Differential and Actionable Awareness: Investigate Abnormal Web Traffic

Our analyst Arthur has been investigating unusual web traffic. In Section 4.3.4, he found suspicious activity and decided to do a deeper dive into the web traffic for his organization. Arthur is expanding his differential awareness by finding more information about traffic that shouldn't be present. His ultimate goal is to provide actionable awareness: find out if the traffic is malicious and make recommendations about how to handle this situation.

Explore Low-packet Flows

As part of this analysis, Arthur decides to explore flows associated with low-volume web sessions. His initial exploration is shown in Example 6.27.

1. Command 1 calls `rwfilter` to pull low-volume flows—that is, flows with 60 or fewer bytes per packet (`--bytes-per=0-60`)—for one particular day. These flows are going to typical web destination ports (`--dport=80,8080,443,8443`). It then pipes the flows to `rwuniq` to profile them by destination port and detected application (`--fields=dport,application`). With this few bytes, it is reasonable that most flows are categorized as application 0, indicating that no application is identified. The flows with identified applications are all reasonable as web protocols.

2. Similarly, Command 2 looks at low-volume flows coming from typical web source ports (`--sport=80,8080,443,8443`) and profiles them according to source port and application. However, these results found traffic coming from port 8080 that is identified as application 53, which is DNS. Given that 8080 is typically a web port, DNS traffic is unusual.

3. Command 3 pivots from this observation to explore how extensive this behavior is on the network. By using both `--sport` and `--dport` of `8080,53`, the call to `rwfilter` will find all traffic that goes between any combination of those two ports. In theory, this would also include traffic between port combinations `53-53` and `8080-8080`, but those combinations are not present on this network. The `rwfilter` feeds into a call to `rwuniq`, generating a summary of the addresses combinations involved.

   While the results involve a total of 17 IP addresses, all of the communication centers around two addresses: `192.168.58` and `10.0.20.58`. Both of these hosts communicate with a separate group of addresses, all of which are outside the organization's network.

These results set up the analytical chaining described in Section 6.3.1. Arthur's investigation of low-packet flows on web ports identified two IP addresses that are exhibiting unusual behavior. He can now pivot to investigating these two internal hosts.

Explore Behavior of Internal Hosts

Example 6.28 shows the next iteration of Arthur's analysis: a pivot to exploring the behavior of the two internal hosts.

1. Commands 1 and 2 investigate activity for `192.168.20.58`. They profile the outbound and inbound network traffic for this IP address according to detected applications and count flows together with distinct endpoints, source ports, and destination ports. The results indicate that this host is a likely DNS resolver. The small number of HTTPS connections reported may be somewhat suspicious.

2. Commands 3 and 4 repeat the same profiling for `10.0.20.58`. The results support that this host is a DNS resolver, slightly stronger than for `192.168.20.58`.

Explore Behavior of Communicating Addresses

Example 6.29 shows a further iteration of Arthur's pivot to exploring the behavior of addresses communicating to these two hosts.

1. Command 1 uses `rwfilter` with `--scidr` to pull all flows sourced from the two addresses. Notice the use of `/32` after each address, which puts it CIDR format so that the two addresses can be listed in the same command. The flows are passed to the `rwset` command to build the set of destination addresses.

```
<1>$ rwfilter --start=2015/06/17 --end=2015/06/17 \
   --type=in,inweb,out,outweb --bytes-per=0-60 \
   --dport=80,8080,443,8443 --pass=stdout \
| rwuniq --fields=dport,application --values=flows,bytes \
   --sort
dPort|appli|   Records|               Bytes|
   80|    0|    830798|           452867494|
   80|   80|     12846|            77938213|
  443|    0|   1175844|           284695302|
  443|   80|        42|              230436|
  443|  443|         2|                9182|
 8080|    0|     62810|            28869568|
 8080|  443|         1|               11418|
 8443|    0|      8331|              343600|
<2>$ rwfilter --start=2015/06/17 --end=2015/06/17 \
   --type=in,inweb,out,outweb --bytes-per=0-60 \
   --sport=80,8080,443,8443 --pass=stdout \
| rwuniq --fields=sport,application --values=flows,bytes \
   --sort
sPort|appli|   Records|               Bytes|
   80|    0|    414610|            39341614|
   80|   80|      1827|              167266|
  443|    0|   1235864|           258751653|
  443|   80|      1139|              543215|
  443|  443|        26|                4060|
 8080|    0|     63918|             9026858|
 8080|   53|         1|                  60|
 8443|    0|      2271|              202872|
<3>$ rwfilter --start=2015/06/17 --end=2015/06/17 \
   --type=in,out,inweb,outweb --sport=8080,53 \
   --dport=8080,53 --pass=stdout \
| rwuniq --fields=sip,dip --values=flows,bytes --sort
            sIP|            dIP|   Records|               Bytes|
     10.0.20.58|    192.5.5.241|         1|                 170|
     10.0.20.58|  192.58.128.30|         1|                 148|
     10.0.20.58|   192.112.36.4|         1|                 170|
     10.0.20.58| 192.203.230.10|         2|                 340|
     10.0.20.58|     198.41.0.4|         1|                 146|
     10.0.20.58|    199.7.83.42|         3|                 488|
     10.0.20.58|   202.12.27.33|         2|                 326|
     67.215.0.5|  192.168.20.58|         4|                 338|
  192.168.20.58|     67.215.0.5|         6|                 676|
  192.168.20.58|    128.8.10.90|         4|                 300|
  192.168.20.58|    128.63.2.53|         7|                 666|
  192.168.20.58|    192.33.4.12|         6|                 672|
  192.168.20.58|  192.36.148.17|         3|                 340|
  192.168.20.58| 192.203.230.10|         3|                 288|
  192.168.20.58| 192.228.79.201|         3|                 288|
  192.168.20.58|   193.0.14.129|         6|                 676|
```

Example 6.27: Isolating Low-byte Web Flows with `rwfilter`

```
<1>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --saddress=192.168.20.58 --pass=stdout \
| rwuniq --fields=application \
   --values=flows,distinct:dip,distinct:sport,distinct:dport \
   --sort
appli|   Records|dIP-Distin|sPort|dPort|
    0|   3708418|        19|64514|15224|
   53|    366479|        18|63293|12143|
   80|         4|         1|    4|    1|
  443|        63|         1|   63|    1|
<2>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --daddress=192.168.20.58 --pass=stdout \
| rwuniq --fields=application \
   --values=flows,distinct:sip,distinct:sport,distinct:dport \
   --sort
appli|   Records|sIP-Distin|sPort|dPort|
    0|   1055265|         8|15297|55579|
   53|    147509|         5|12143|28624|
   80|         4|         1|    1|    4|
  137|         6|         2|    1|    1|
  443|        63|         1|    1|   63|
<3>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --saddress=10.0.20.58 --pass=stdout \
| rwuniq --fields=application \
   --values=flows,distinct:dip,distinct:sport,distinct:dport \
   --sort
appli|   Records|dIP-Distin|sPort|dPort|
    0|   1457893|        22|64514|14184|
   53|      6779|         1|    1| 2366|
<4>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --daddress=10.0.20.58 --pass=stdout \
| rwuniq --fields=application \
   --values=flows,distinct:sip,distinct:sport,distinct:dport \
   --sort
appli|   Records|sIP-Distin|sPort|dPort|
    0|    326762|         8|14194|  820|
   53|      6789|         1| 2367|    1|
  137|         2|         1|    1|    1|
```

Example 6.28: Pivoting with `rwfilter` and `rwuniq` to Explore Endpoint Behavior

2. Command 2 follows the same process to build the set of source addresses for flows that have either of the two addresses as destinations.

3. Command 3 uses `rwsettool --union` to perform a union of the two sets. This produces a set of contact IP addresses.

4. Command 4 uses `rwsetcat --count` to count the number of IP addresses in the set—26 addresses in all.

5. The traffic for these addresses is then profiled in Commands 5 (source) and 6 (destination). These commands look at the flows by application and protocol, counting flows, distinct endpoint addresses, distinct source and destination ports, and distinct byte lengths.

   The two profiles match pretty closely, except for the Microsoft NETBIOS Name Service (UDP 137) flows. There are over double the number of flows sourced from the contact addresses than destined to those addresses.

6. To explore the difference, Arthur looks at addresses reserved by the internet protocol as shown in Command 7. The reserved addresses are the minimum address per CIDR/24 block ending in `.0` (the address for the block as a whole) and the maximum address per CIDR/24 block ending in `.255` (the broadcast address going to each host in that block).

   Neither of these addresses should be used for name service traffic. However, by using `rwfilter` and `rwuniq`, Arthur found one address (`192.168.40.201`) that is sending almost enough flows to two of these addresses to constitute the identified difference.

Conclusions from the Exploratory Analysis for Situational Awareness

Arthur used the exploratory analysis technique of analytical chaining to navigate through a chain of clues and isolate traffic of interest. His analysis identified hosts that are attempting to bypass traffic controls (by using port 8080 to source DNS queries) and scan the network for name services (by addressing the net and broadcast addresses). These hosts must be secured to halt this activity.

While Arthur did not find evidence of web redirection, he did find actionable evidence of suspicious name service traffic. His results identified potential security issues for the organization's name resolution infrastructure and his recommendations will help with efforts to remediate these problems.

```
<1>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --scidr=10.0.20.58/32,192.168.20.58/32 --pass=stdout \
| rwset --dip-file=dip-contact.set
<2>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --dcidr=10.0.20.58/32,192.168.20.58/32 --pass=stdout \
| rwset --sip-file=sip-contact.set
<3>$ rwsettool --union sip-contact.set dip-contact.set \
   --output=contact.set
<4>$ rwsetcat --count-ips contact.set
26
<5>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --sipset=contact.set --application=1-79,81-442,444- \
   --pass=stdout \
| rwuniq --fields=application,protocol \
   --values=flows,distinct:sip,distinct:dip,distinct:dport,distinct:sport,distinct:bytes \
   --sort
appli|pro|   Records|sIP-Distin|dIP-Distin|dPort|sPort|bytes-Dist|
   53| 17|    401357|         5|        85|38248|15351|       807|
  137| 17|      1052|         3|        75|    1|  408|        45|
  138| 17|        46|         1|         1|    1|    1|         1|
  139|  6|      4239|         2|        93| 2023| 1291|       332|
  389|  6|      4272|         2|        89| 2376|    2|       431|
  389| 17|      3186|         3|        93| 2135|    6|        11|
 5004| 17|         2|         2|         2|    2|    2|         1|
<6>$ rwfilter --start=2015/06/17 --end=2015/06/17 --type=all \
   --dipset=contact.set --application=1-79,81-442,444- \
   --pass=stdout \
| rwuniq --fields=application,protocol \
   --values=flows,distinct:sip,distinct:dip,distinct:dport,distinct:sport,distinct:bytes \
   --sort
appli|pro|   Records|sIP-Distin|dIP-Distin|dPort|sPort|bytes-Dist|
   53| 17|    685348|        84|        18|15347|63627|       835|
  137| 17|       511|        66|         3|    1|    1|        16|
  138| 17|       265|        13|         1|    1|    1|         3|
  139|  6|      4239|        93|         2| 1291| 2023|       605|
  389|  6|      4272|        89|         2|    2| 2376|       657|
  389| 17|      3553|        93|         3|    6| 2447|        39|
 5004| 17|         2|         2|         2|    2|    2|         1|
<7>$ rwfilter --start=2015/06/17 --end=2015/06/17 \
   --type=all --application=137 --sipset=contact.set \
   --daddress=x.x.x.0,255 --pass=stdout \
| rwuniq --fields=sip --values=flows,distinct:dip
           sIP|   Records|dIP-Distin|
  192.168.40.20|       507|         2|
```

Example 6.29: Pivoting with `rwfilter` to Explore Contacts of Endpoints

6.4 Summary of SiLK Commands in Chapter 6

Command	Section Name	Page
`rwfilter`	Using Tuple Files for Complex Filtering	134
	Using Prefix Maps to Filter Flow Records with `rwfilter`	158
`rwbagtool`	Adding and Subtracting Bags	137
	Multiplying and Dividing Bags	139
	Thresholding Bags with Count and Key Parameters	141
`rwcat`, `rwappend`	Combining Flow Record Files to Provide Context	143
`rwsplit`	Dividing and Sampling Flow Record Files with `rwsplit`	145
`rwtuc`	Generate Flow Records From Text	147
`rwaggbag`	Creating Aggregate Bags from Flow Records	149
`rwaggbagbuild`	Creating Aggregate Bags from Text	151
`rwaggbagcat`	Viewing Aggregate Bags	150
`rwaggbagtool`	Thresholding Aggregate Values	151
	Exporting Aggregate Bags to Bags and IPsets	153
`rwuniq`	Comparing `rwaggbag` with `rwuniq`	151
`rwpmapbuild`	Creating a User-defined Prefix Map From a Text File	155
`rwcut`	Displaying Prefix Labels with `rwcut`	159
`rwgroup`, `rwsort`, `rwstats`, `rwuniq`	Sorting, Counting, and Grouping Records with Prefix Maps	160
`rwpmaplookup`	Querying Prefix Map Labels	161

This page intentionally left blank.

Chapter 7

Case Studies: Advanced Exploratory Analysis

This chapter features two detailed case studies of exploratory analysis, using concepts from previous chapters. Both employ the SiLK workflow, SiLK tools, UNIX commands, and networking concepts to provide a practical example of exploratory analyses with network flow data.

Upon completion of this chapter you will be able to

- describe how to use single-path and multi-path analyses as the building blocks of an exploratory analysis
- formulate an approach to exploratory analysis
- prepare a model of exploratory analysis
- execute these analyses with various SiLK tools in one automated program

The case studies build upon the answers to these questions to investigate unusual network traffic and revealing changes of network behavior.

7.1 Dataset for Exploratory Case Studies

Like the previous case studies and command examples, the exploratory analysis case studies uses the FCCX dataset described in Section 1.8. From the diagram in Figure 1.4, we know that sensors `S0` through `S4` monitor the operating network. These sensors are part of the inventory generated via the example in Chapter 5.

7.2 Case Study: Investigating Suspicious TCP Behavior

Omar is a security administrator who is investigating TCP traffic for signs of illicit behavior. His wants to explore TCP requests during a specific time period to figure out which ones are legitimate and which ones are not. This will identify source and destination IP addresses for further investigation. His overall goal is to figure out whether a pattern of activity is associated with this traffic—and if so, what it might indicate.

Omar's exploratory analysis follows the analysis workflow from Section 1.5. Each level of his analysis is posed as a question that he needs to answer.

7.2.1 Level 0: Which TCP Requests are Suspicious?

In the initial phase of Omar's exploratory analysis, he would like to identify which TCP requests might represent illegitimate traffic. When he starts his analysis, he has no idea as to which flows are legitimate and which ones are not. However, he does know that, for most services, the flows containing client requests tend to be close in number to those containing responses to those requests. He will exploit this tendency to identify irregular traffic.

In particular, Omar wants to look at traffic on service ports to find out which ports carry a much higher volume of inbound data than outbound data. This is one of the fingerprints (or indicators) of network scanning and several other behaviors that may be of concern. Scanning usually involves low data volume (bytes) but high numbers of packets or requests (which would equate to flows).

Although this traffic imbalance is outside the range of typical behavior, it may not represent malicious activity. Omar needs to identify these ports to take a closer look at their TCP traffic and assess any impact it might have.

Example 7.1 shows how Omar uses SiLK to find this type of data anomaly. His analytic will retrieve inbound and outbound TCP requests on all network ports. It then finds the ports that have a much higher level of inbound requests than outbound requests.

1. Command 1 uses `rwfilter` to pull inbound TCP (protocol 6) traffic for sensors `S0` through `S4` sent to all reserved ports (0 - 1023), which are dedicated to network services. It pipes the results to the `rwuniq` command, which counts flows and bytes for each destination port, then sorts the results to present the ports in ascending order.

2. Similarly, Command 2 uses `rwfilter` to pull outbound TCP traffic and `rwuniq` to count and sort traffic for each source port.

Omar compares the corresponding ports in the results for Commands 1 and 2 in Example 7.1. He notices that most of the service ports carry similar levels of inbound and outbound TCP traffic. However, ports 21, 22, and 591 carry higher inbound than outbound TCP traffic. He decides to further investigate activity on these ports.

```
<1>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=in --proto=6 --dport=0-1023 \
   --pass=stdout \
| rwuniq --fields=dport --values=flows,bytes --sort
dPort|   Records|               Bytes|
   21|      6184|              418452|
   22|      6180|              481760|
   53|        35|               88237|
   88|     47187|            74586782|
  135|      6996|             6940590|
  137|      6064|              364320|
  139|     57313|            93274792|
  389|     22095|           118221682|
  445|     81039|           363318703|
  591|    112702|            76762887|
<2>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=out --proto=6 --sport=0-1023 \
   --pass=stdout \
| rwuniq --fields=sport --values=flows,bytes --sort
sPort|   Records|               Bytes|
   21|       200|               28288|
   22|       204|               99024|
   53|        35|               18860|
   88|     47184|            76069648|
  135|      6991|             4052306|
  137|        80|                3200|
  139|     51631|            47100178|
  389|     22214|           110222710|
  445|     75426|           205229733|
  591|     55161|            17580192|
```

Example 7.1: Looking for Service Ports with Higher Inbound than Outbound TCP Traffic

7.2.2 Level 1: Which Requests are Illegitimate?

The nest step in Omar's exploratory analysis is to separate normal and abnormal TCP requests on the service ports identified in Section 7.2.1. Specifically, he wants to identify mismatched TCP flows: flows that have either no request or no response. These flows are odd, and worth examining to see if they are malicious. His goal is to describe this behavior in a way that supports further analysis.

Example 7.2 shows how Omar detects mismatched flows, identifies the IP addresses of their sources, and contrasts those addresses with sources of matched flows.

1. Command 1 uses `rwfilter` to pull inbound TCP traffic for sensors `S0` through `S4` on the suspect destination ports 21, 22, and 591. It then uses `rwsort` to sort this traffic by source IP address, destination port, destination IP address, source port, protocol, and start time. The sorted inbound flow records are saved in the file `app-in.raw`. This sort order sets up the data for matching in Command 3.

2. Similarly, Command 2 uses `rwfilter` to pull outbound TCP traffic for the sensors and ports of interest. It sorts this traffic by destination IP address, source port, source IP address, protocol, and start time. The sorted outbound flow records are saved in the file `app-out.raw`. This sort order allows records to be matched efficiently with those from Command 1, using `rwmatch`.

3. To find mismatched flows, Command 3 uses `rwmatch` to match queries in the inbound TCP flows in `app-in.raw` to responses in the outbound TCP flows in `app-out.raw`. Mismatched flows that do not belong to sessions can indicate illegitimate activity.

   - `--relate=1,2` matches the inbound source IPs to the outbound destination IPs.
   - `--relate=2,1` matches the inbound destination IPs to the outbound source IPs.
   - `--relate=3,4` matches the inbound source ports to the outbound destination ports.
   - `--relate=4,3` matches the inbound destination ports to the outbound source ports.
   - `--relate=5,5` matches the inbound and outbound protocols.
   - `--unmatched=b` saves the unmatched inbound and outbound records instead of discarding them. These unmatched records are the ones we will want to investigate for illegitimate behavior.

   The matched records are indicated by setting the (mostly unused) next-hop IP address. Rather than a real IP address, `rwmatch` uses 0 followed by a positive integer value for a request, and 255 followed by the corresponding integer for a response. For a request without a response, `rwmatch` uses `0.0.0.0` for a response without a request, `rwmatch` uses `255.0.0.0`. Command 3 then calls `rwsort` to sort the matched records by start time and saves them in a temporary file, `temp-match.raw`.

4. Command 4 runs `rwfilter` on `temp-match.raw` to filter records that have a next-hop IP indicating unmatched inbound requests, then saves these records in `temp-noresp.raw`. A second call to `rwfilter` filters records that have a next-hop IP indicating unmatched outbound responses, then saves those records in `temp-noreq.raw`. The records that fail both filters represent flows with matching responses and are saved in `app-match.raw`.

5. Command 5 runs the `rwstats` command on `temp-noresp.raw` to display information about the top five source IP addresses and destination ports of flows with no response flows. Since there are only two source IP addresses in this data (one with two destination ports), only three are displayed.

6. Command 6 runs the `rwstats` command on `app-match.raw` to display information about the top five source IP addresses and source ports of flows that do have matching query and response flows. Since there are 3,848 source IP addresses for the matching records, the top five are shown. None of these are the sources for the unmatched records.

Given the distribution of source addresses between the matched and unmatched traffic, Omar decides that these sources are clearly worth investigating further.

```
<1>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=in --proto=6 --dport=21,22,591 \
   --pass=stdout \
| rwsort --fields=sip,dport,dip,sport,protocol,stime \
   --output=app-in.raw
<2>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=out --proto=6 --sport=21,22,591 \
   --pass=stdout \
| rwsort --fields=dip,sport,sip,dport,protocol,stime \
   --output=app-out.raw
<3>$ rwmatch --relate=1,2 --relate=2,1 --relate=3,4 --relate=4,3 \
   --relate=5,5 --unmatched=b app-in.raw app-out.raw stdout \
| rwsort --fields=stime --output=temp-match.raw
<4>$ rwfilter temp-match.raw --next=0.0.0.0 \
   --pass=temp-noresp.raw --fail=stdout \
| rwfilter stdin --next=255.0.0.0 --pass=temp-noreq.raw \
   --fail=app-match.raw
<5>$ rwstats --fields=sip,dport --values=flows --count=5 \
   temp-noresp.raw
INPUT: 69810 Records for 3 Bins and 69810 Total Records
OUTPUT: Top 5 Bins by Records
            sIP|dPort|   Records|  %Records|   cumul_%|
     10.0.40.21|  591|     57582| 82.483885| 82.483885|
  192.168.181.8|   22|      6180|  8.852600| 91.336485|
  192.168.181.8|   21|      6048|  8.663515|100.000000|
<6>$ rwstats --fields=sip,sport --values=flows --count=5 \
   app-match.raw
INPUT: 110478 Records for 3848 Bins and 110478 Total Records
OUTPUT: Top 5 Bins by Records
            sIP|sPort|   Records|  %Records|   cumul_%|
192.168.165.216|  591|      1708|  1.546009|  1.546009|
192.168.161.124|  591|      1691|  1.530621|  3.076631|
192.168.122.195|  591|      1663|  1.505277|  4.581908|
192.168.122.141|  591|      1657|  1.499846|  6.081754|
 192.168.121.57|  591|      1657|  1.499846|  7.581600|
```

Example 7.2: Identifying Abnormal TCP Flows and their Originating Hosts

7.2.3 Level 2: What are the Illegitimate Sources and Destinations Doing?

In the next part of Omar's exploratory analysis, he will investigate the activities of the illegitimate source and destination hosts identified in Section 7.2.2 to see what patterns emerge.

Level 2A: What are the Illegitimate Source IPs Doing?

Omar first decides to take a look at the activity of the illegitimate source IP addresses. As shown in Example 7.3, he will find the sources of illegitimate traffic and look at the network behavior associated with scanning. Scan queries typically have low byte counts and often lack corresponding responses, since the scanned hosts lack the service being sought.

1. Command 1 calls `rwbag` and `rwbagtool --coverset` to create a set of source IPs that did not have matching queries (`noresp.set`). As input, it uses the file of no-response flows created in Section 7.2.2 (`temp-noresp.raw`).
2. Command 2 calls `rwfilter` to pull records coming from the source IP addresses `noresp.set`—in other words, records with source IPs that produced unmatched flows. It saves these records in `sources.raw`.
3. To find the actual scanning flows, Command 3 uses `rwfilter` on `sources.raw` to filter flows with very low byte counts (0-60 bytes), indicating those with only a header, or with a header and optional extensions. It then uses the `rwuniq` command to profile these flows and saves the results in `source.txt`. This file is human readable; it contains a breakdown by protocol and bytes per flow that shows how many hosts were the destination of these flows, when the earliest of them started, and when the latest ended.
4. A copy of the low-volume flows is sent to a second call to `rwuniq` to save the earliest start and latest end times of the flows from each source IP to the file `source-fields.txt`. The output here is generated without column headings and vertical bar delimiter to facilitate processing by commands 5-9. As it happens, all of these low-volume flows come from a single source IP address, so the `source-fields.txt` only contains one line.
5. Command 4 displays the contents of the human-readable profile. There are several features in this output:
   - There are few distinct byte sizes in these results: only one for TCP flows, and only two for UDP flows.
   - The TCP flows are by far the most numerous, and go to by far the most distinct addresses.
   - The earliest start times are all within a one minute range.
   - The latest end times are all within a five second span.

Based on these results, Omar interprets this behavior as scan traffic with a common direction. That all of these flows come from a single source IP address further supports his interpretation.

Level 2B: What Behavior Changes do Destination IPs Show?

Next, Omar investigates traffic patterns on the destination hosts (the targets of the scans). Using the start times and end times of the scans from Example 7.3 as starting points, he will look at traffic patterns before and afterwards as shown in Example 7.4.

1. Commands 1 through 5 locate the start times (`StTime`, `StEpoch`) and end times (`EnTime`, `EnEpoch`), using the `source-fields.txt` file created in Example 7.3. This identifies the "before" and "after" boundaries of the scan.

```
<1>$ rwbag --bag-file=sipv4,flows,stdout temp-noresp.raw \
| rwbagtool --coverset --output=noresp.set
<2>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=in,out --sipset=noresp.set \
   --pass=sources.raw
<3>$ rwfilter sources.raw --bytes=0-60 --pass=stdout \
| rwuniq --fields=protocol,bytes \
   --values=flows,distinct:dip,stime-earliest,etime-latest \
   --sort --output=sources.txt --copy=stdout \
| rwuniq --fields=sip --values=stime-earliest,etime-latest \
   --no-titles --delim=' ' --output=source-fields.txt
<4>$ cat sources.txt
pro|     bytes|   Records|dIP-Distin|     sTime-Earliest|       eTime-Latest|
  6|        60|     29492|       768|2015/06/17T16:12:58|2015/06/17T16:41:21|
 17|        29|        68|        17|2015/06/17T16:13:54|2015/06/17T16:41:21|
 17|        42|       136|        17|2015/06/17T16:13:54|2015/06/17T16:41:26|
<5>$ srcArray=( $(cat source-fields.txt) )
```

Example 7.3: Finding Activity of Illegitimate Destination IP Addresses

(a) Command 1 stores the contents of the file (one line) in a shell array for ease of reference to the fields.

(b) Command 2 converts the earliest start time of the scan into an integer UNIX epoch value (the number of seconds since midnight, January 1, 1970). This format is used to make it easy to calculate.

(c) Command 3 subtracts one from the epoch value to get a time briefly before the scan activity started, then uses the string processing language `awk` to convert the epoch date back into a SiLK formatted date.

(d) Commands 4 and 5 do an analogous process to commands 2 and 3, but with the ending time of the scanning activity, producing a value just after the scanning ended.

2. Command 6 uses `rwfilter` to pull inbound and outbound non-web traffic for the destination IPs in `noresp.set` in the time window *before* the start of the scan, saving these flows to `dest-before.raw`.

3. Command 7 uses `rwfilter` to pull inbound and outbound traffic for the illegitimate IPs in `noresp.set` in the time window *after* the end of the scan, saving these files to `dest-after.rw`.

4. For clarity of display, Command 8 calls `rwuniq`, then uses the `head` command to pull off the column headers and store them in the file `myhead`.

5. After displaying the column headers with `cat`, Command 9 uses `rwuniq` to calculate the counts of flows for each byte size in `dest-before.rw`, which contains data from the time period before the start of scanning. It sorts the byte sizes in ascending order, then calls `tail` to display the largest flows. (Because Omar used `tail` for this, he had to separately save and display the column headers; without that, no column labels would be shown.)

6. To profile network behavior after scanning, Command 10 uses `rwuniq` to find the flows with the largest byte sizes in the file `dest-after.rw`, which contains data from the time period after scanning ended.

```
<1>$ srcArray=( $(cat source-fields.txt) )
<2>$ StEpoch=$( date -d ${srcArray[1]} +"%s")
<3>$ StTime=$(echo $(( $StEpoch - 1 )) | awk '{print \
   strftime("%Y/%m/%dT%T",$1)}')
<4>$ EnEpoch=$( date -d ${srcArray[2]} +"%s")
<5>$ EnTime=$(echo $(( $EnEpoch + 1 )) | awk '{print \
   strftime("%Y/%m/%dT%T",$1)}')
<6>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=in,out --dipset=noresp.set \
   --etime=2015/06/01-2015/06/17T09:12:57 \
   --pass=dest-before.raw
<7>$ rwfilter --sensor=S0,S1,S2,S3,S4 --start=2015/06/01 \
   --end=2015/06/30 --type=in,out --dipset=noresp.set \
   --stime=2015/06/17T09:41:27-2015/06/30 \
   --pass=dest-after.raw
<8>$ rwuniq --fields=bytes,protocol \
   --values=Flows,distinct:sip,distinct:dip dest-before.raw \
   | head -1 >myhead
<9>$ cat myhead; \
rwuniq --fields=bytes,protocol \
   --values=Flows,distinct:sip,distinct:dip --sort \
   dest-before.raw \
| tail -5
     bytes|pro|   Records|sIP-Distin|dIP-Distin|
       884| 17|         2|         1|         1|
       988|  1|         4|         1|         1|
      1028| 17|         5|         2|         1|
      1326| 17|         1|         1|         1|
      1976|  1|         3|         1|         1|
<10>$ cat myhead; \
rwuniq --fields=bytes,protocol \
   --values=Flows,distinct:sip,distinct:dip --sort \
   dest-after.raw \
| tail -5
     bytes|pro|   Records|sIP-Distin|dIP-Distin|
      5128|  1|         8|         2|         1|
      5488|  1|         4|         1|         1|
      5756|  1|         4|         1|         1|
      5844|  1|         4|         1|         1|
      6020|  1|         4|         1|         1|
```

Example 7.4: Finding Changed Behavior in Destination IPs

7.2.4 Level 3: What are the Commonalities Across The Cases?

Omar's exploratory analysis of the network traffic shows likely scanning behavior during a tight time frame: a 28-minute interval, ending within a few seconds.

A further look at the results reveals a definite change in behavior of the suspected scanners. One IP address is active before and during the scan. Another is active after the scan, with much larger flows to the destination after the scan completed. All of this is highly suspicious. Even more concerning, the large traffic after the scan is all ICMP traffic (which is normally quite modest in size). These large ICMP flows are highly unusual for any benign purpose.

Omar concludes that his exploration of the TCP data has uncovered illicit behavior that requires a response from his organization. He identifies several areas for future investigation, including

- looking for additional hosts that exhibit behavior that is similar to the scan/exploit hosts, during the periods before and after the scan
- investigating the behavior of the scan and exploit hosts for further confirmation of their malicious character
- looking for other activities associated with the scan/exploit hosts. What else are they up to?

These areas will likely suggest others to be explored.

One difficulty in exploratory analysis is knowing when to stop! Omar needs to keep the desired level of output firmly in mind and direct his explorations towards providing suitable results. He needs to stop either when those results are found (i.e., a likely compromise is identified, a sufficient understanding of the service has resulted, or interactions across the network are understood), or when it is clear no such results will emerge (i.e., everything is benign).

In this case, Omar identified a definite change in behavior that shows signs of being malicious. He therefore decides to wrap up his exploratory analysis. He can now report information about the affected hosts to incident handlers and system administrators for response. He can also pass on his suggestions for future areas of investigation into this incident.

7.3 Case Study: Exploring Network Messaging for Information Exposure

Beth is an enterprise security operations analyst who's been tasked with investigating information exposure. (For the purpose of this case study, *information exposure* is defined as the inadvertent exposure of network information due to relaxed security controls that are commonly exploited by attackers.) She will apply the exploratory analysis concepts from Chapter 6 and her knowledge of networking to investigate how use of the Internet Control Message Protocol (ICMP) can expose internal information.

7.3.1 Prepare a Model of the Enterprise Network and Protocols

As described in RFC 792[20], ICMP is a crucial component of the Internet Protocol (IP) for reporting errors in the processing of datagrams. Unfortunately, when relaxed security controls are implemented in a network, ICMP can leak information to attackers as they conduct network reconnaissance. For example, subnetwork gateways commonly respond to ICMP echo request messages (ping) with an echo response.

However, as the network perimeter continues to shrink due to technologies such as mobile devices and the Internet of things (IoT), hosts within a subnetwork should not respond to ICMP pings outside of their local subnetwork. If they do respond, attackers are able to gather information that supports their efforts.

By analyzing ICMP network flow data, Beth can find the ground truth of how an enterprise network is configured and operates. This will help her to identify security controls that may require further review.

Build a Port-Protocol Prefix Map

The first step in Beth's exploratory analysis is to define a model of the enterprise network and the protocols under analysis. She will use SiLK IPsets and prefix maps (pmaps) for this modeling. These data structures can be created by using two pieces of information: the network diagram from Figure 1.4 and the network's sensor configuration file, `sensor.conf`. RFC 792 provides some of the protocol information required for the model. Beth must research and gather the remaining information.

To begin, Beth builds a port-protocol prefix map (also known as a proto-port pmap) of the ICMP protocol as defined by RFC standards. Example 7.5 shows her pmap and how it is created.

1. She first creates a text file with protocol-port mappings, `icmp_code_official_pmap.txt`, as shown in Command 1. Each entry in this text file defines an ICMP type, code, and a corresponding text label.

   By design, SiLK records do not store ICMP types and codes in individual flow record fields. Instead, an algorithm calculates the type and code and stores it in the flow record `dPort` field. The algorithm used for this is `(type x 256) + code`. For example, the prefix map entry of ICMP type 3 code 0 (network unreachable) is protocol 1, port 768 (3 x 256 + 0).

2. She then converts the text file into a binary pmap for use with SiLK tools, as shown in Commands 2 and 3. The binary pmap will be used with other SiLK tools to count and sort records with the protocol and port combinations specified in the pmap.

```
<1>$ cat <<-EOF >icmp_code_official.pmap.txt
map-name icmp
mode proto-port

1/0     1/0     Echo Reply      # type 0 code 0

1/768   1/768   Net Unreachable # type 3 code 0
1/769   1/769   Host Unreachable # type 3 code 1
1/770   1/770   Protocol Unreachable # type 3 code 2
1/771   1/771   Port Unreachable # type 3 code 3
1/772   1/772   Frag Needed DF Set # type 3 code 4
1/773   1/773   Source Rte Failed # type 3 code 5
1/774   1/774   Dest Ntwk Unknown # type 3 code 6
```

[20] Internet Engineering Task Force (IETF) Request for Comments 792 (https://www.ietf.org/rfc/rfc792.txt)

```
1/775   1/775   Dest Host Unknown # type 3 code 7
1/776   1/776   Source Host Isolated # type 3 code 8
1/777   1/777   Dest Ntwk Admin Proh # type 3 code 9
1/778   1/778   Dest Host Admin Proh # type 3 code 10
1/779   1/779   Dest Ntwk Unreachable for TOS # type 3 code 11
1/780   1/780   Dest Host Unreachable for TOS # type 3 code 12
1/781   1/781   Administratively Prohibited # type 3 code 13
1/782   1/782   Host Precedence Violation # type 3 code 14
1/783   1/783   Precedence Cutoff # type 3 code 15

1/1024  1/1024  Source Quench # type 4 code 0 - DEPRECATED

1/1280  1/1280  Redirect Ntwk/Subnet # type 5 code 0
1/1281  1/1281  Redirect Host # type 5 code 1
1/1282  1/1282  Redirect TOS & Ntwk/Subnet # type 5 code 2
1/1283  1/1283  Redirect TOS & Host # type 5 code 3

1/1536  1/1536  Alternate Host Address # type 6 code 0

1/2048  1/2048  Echo request # type 8 code 0

1/2304  1/2304  Normal Router Advertisement # type 9 code 0
1/2320  1/2320  Router Adv not rte common traff # type 9 code 16

1/2560  1/2560  Router Selection # type 10 code 0

1/2816  1/2816  TTL exceeded # type 11 code 0
1/2817  1/2817  Frag Reassembly Time Exceeded # type 11 code 1

1/3072  1/3072  Parameter Prob - ptr indicates err # type 12 code 0
1/3073  1/3073  Missing Required Option # type 12 code 1
1/3074  1/3074  Bad Length # type 12 code 2

1/3328  1/3328  Timestamp request # type 13 code 0
1/3584  1/3584  Timestamp Reply # type 14 code 0
1/3840  1/3840  Information Request # type 15 code 0
1/4096  1/4096  Information Reply # type 16 code 0
1/4352  1/4352  Address Mask Request # type 17 code 0
1/4608  1/4608  Address Mask Reply # type 18 code 0

1/7680  1/7680  Traceroute pkt forwarded # type 30 code 0
1/7681  1/7681  Traceroute no rte for pkt; discarded # type 30 code 1

1/7936  1/7936  DG conv failed - unspecified # type 31 code 0
1/7937  1/7937  DG conv failed - Don't Convert option # type 31 code 1
1/7938  1/7938  DG conv failed - unk mandatory opt # type 31 code 2
1/7939  1/7939  DG conv failed - known unsupported opt # type 31 code 3
1/7940  1/7940  DG conv failed - unsupported transport proto # type 31 code 4
1/7941  1/7941  DG conv failed - length exceeded # type 31 code 5
1/7942  1/7942  DG conv failed - IP hdr len exceeded # type 31 code 6
1/7943  1/7943  DG conv failed - Transport proto > 255 # type 31 code 7
1/7944  1/7944  DG conv failed - Port conv out of range # type 31 code 8
1/7945  1/7945  DG conv failed - Transport hdr len exceeded # type 31 code 9
1/7946  1/7946  DG conv failed - 32bit Rollover missing & ACK set # type 31 code 10
```

```
1/7947  1/7947  DG conv failed - Unk mandatory transport opt # type 31 code 11

1/9472  1/9472  Domain Name Request # type 37 code 0
1/9728  1/9728  Domain Name Reply # type 38 code 0

1/10240 1/10240 Photuris bad SPI # type 40 code 0
1/10241 1/10241 Photuris Auth Failed # type 40 code 1
1/10242 1/10242 Photuris Decompression Failed # type 40 code 2
1/10243 1/10243 Photuris Decryption Failed # type 40 code 3
1/10244 1/10244 Photuris Need Authentication # type 40 code 4
1/10245 1/10245 Photuris Need Authorization # type 40 code 5

1/10496 1/10496 Seamoby CARD or CXTP # type 41 code 0 (RFC 4066/4067)
EOF
<2>$ rwpmapbuild --input-file=icmp_code_official.pmap.txt \
   --output-file=icmp_code_official.pmap
<3>$ file icmp_code_official.pmap.txt icmp_code_official.pmap
icmp_code_official.pmap.txt: ASCII text
icmp_code_official.pmap:     SiLK, PREFIXMAP v3, Little Endian, Uncompressed
```

Example 7.5: Building an RFC Compliant ICMP Type and Code Prefix Map

Define IP Address Roles

Beth's next step in creating the model is to define IP address roles. She will look at IP addresses that are associated with internal subnetworks, internal subnetwork gateways, and network services. Example 7.6 shows how she defines all of the internal subnetworks contained in `sensor.conf`.

1. The file `internal.set.txt` (shown in Command 1) lists the IP addresses of the internal networks from `sensor.conf`.
2. Command 2 calls the `rwsetbuild` command to convert this text list into a binary SiLK IPset, `internal.set` (shown in Command 3).

Beth uses the resulting IPset, `internal.set`, to filter hosts the sensor identifies as "internal" to the monitored network from hosts the sensor identifies as "external" to the network.

```
<1>$ cat <<-EOF >internal.set.txt
# Internal networks from FCCX-15 sensor.conf
10.0.x.x
192.168.x.x
EOF
<2>$ rwsetbuild internal.set.txt >internal.set
<3>$ file internal.set.txt internal.set
internal.set.txt: ASCII text
internal.set:     SiLK, IPSET v2, Little Endian, LZO compression
```

Example 7.6: Building an IPset of FCCX-15 Internal Subnetworks

She then uses the `rwsetbuild` command to build IPsets that model internal subnetwork gateways (`gateways.set`, shown in Example 7.7) and network services (`services.set`, show in Example 7.8). The gateway and services information comes from the network diagram in Figure 1.4.

```
<1>$ cat <<-EOF >gateways.set.txt
# FCCX-15 gateways commonly end in .1 or .254
10.0.x.1
10.0.x.254
192.168.x.1
192.168.x.254
EOF
<2>$ rwsetbuild gateways.set.txt >gateways.set
<3>$ file gateways.set.txt gateways.set
gateways.set.txt: ASCII text
gateways.set:     SiLK, IPSET v2, Little Endian, LZO compression
```

Example 7.7: Building an IPset of FCCX-15 Internal Subnetwork Gateways

```
<1>$ cat <<-EOF >services.set.txt
# Addresses of FCCX-15 internal network services
10.0.20.x # DMZ subnet
10.0.40.x # Services subnet
192.168.20.x # DMZ subnet
192.168.40.x # Services subnet
EOF
<2>$ rwsetbuild services.set.txt >services.set
<3>$ file services.set.txt services.set
services.set.txt: ASCII text
services.set:     SiLK, IPSET v2, Little Endian, LZO compression
```

Example 7.8: Building an IPset of FCCX-15 Internal Network Service Subnetworks

7.3.2 Pull Records Associated with ICMP Flows

After defining the network model, Beth uses the `rwfilter` command to pull ICMP traffic from the repository as shown in Example 7.9.

1. Command 1 pulls the ICMP data (`--proto=1`) and saves it to the file `icmpall.rw`. The summary statistics (`--print-statistics`) show that 436,648 records in the repository matched the query.
2. Command 2 shows the resulting SiLK raw file is less than 5 Mb in size.

This shows the amount of traffic that she must investigate.

Hint 7.1: Limiting `rwfilter` Query Size

If you are unsure about the number of records that might result from an `rwfilter` query, remove the `--pass=icmpall.rw` option from the `rwfilter` command. This displays information about the query without creating the raw file, allowing you to preview its results.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
```

```
   --type=all --protocol=1 --pass=icmpall.rw \
   --print-statistics
Files  2442.  Read   39210497.  Pass     436648. Fail    38773849.
<2>$ ls -ahl icmpall.rw
-rw-r--r--. 1 analyst analyst 4.9M Feb 27 15:42 icmpall.rw
```

Example 7.9: Pulling ICMP Network Flow Data for a Specified Period

7.3.3 What Anomalies are in the ICMP Records?

To find potential cases of data exposure, Beth will now examine the ICMP records for anomalies. SiLK's `rwuniq` command is a good tool for reviewing and examining data that's pulled from the repository. `rwuniq` provides an aggregate view of SiLK flow records. It can be used with prefix maps to label fields and display a visual summary of ICMP messaging.

As shown in Command 1 of Example 7.10, Beth combines `rwuniq` with the prefix map from Section 7.3.1 to generate a sorted summary of ICMP types, codes, distinct source/destination IP address counts, and flow record counts.

- `--pmap-file=icmp:icmp_code_official.pmap` tells `rwuniq` to count and summarize flow records according to the labels defined in the pmap file from Example 7.5.
- `--fields=itype,icode,dst-icmp` identifies the flow fields that `rwuniq` selects and sorts upon. `itype` and `icode` represent the ICMP type and code computed for the flow record. `dst-icmp` is the record's label from the pmap.
- `--values=sip-distinct,dip-distinct` counts the distinct source and destination IP addresses for each pmap label.
- `icmpall.rw` contains the ICMP flow data (created in Example 7.9)

The `UNKNOWN` labels in the results from Example 7.10 are a default feature of prefix maps. They are assigned to any attributes that do not have a corresponding pmap entry. This feature helps to identify ICMP protocol anomalies in flow data. The results from the `rwuniq` command contain ICMP echo reply anomalies (type (`iTy`) `0`, code (`iCo`) `1` and type `0`, code `9`) and ICMP echo request anomalies (type `8`, code `1` and type `8`, code `9`). Internet research[21,22] indicates that these anomalies, when combined with the other ICMP types and codes displayed in the `rwuniq` summary, indicate a high probability of network reconnaissance in the data.

```
<1>$ rwuniq --pmap-file=icmp:icmp_code_official.pmap \
   --fields=itype,icode,dst-icmp \
   --values=sip-distinct,dip-distinct --flows --sort-output \
   icmpall.rw
iTy|iCo|            dst-icmp|sIP-Distin|dIP-Distin|   Records|
  0|  0|          Echo Reply|       106|        70|    101076|
  0|  1|             UNKNOWN|        13|        18|        39|
  0|  9|             UNKNOWN|        11|         1|        58|
  3|  0|     Net Unreachable|         3|         6|      1074|
```

[21] https://nmap.org/book/osdetect-methods.html#idm8844
[22] https://www.blackhat.com/presentations/bh-europe-00/OfirArkin/OfirArkin2.pdf

```
  3|   1|     Host Unreachable|         12|         78|     142054|
  3|   2|Protocol Unreachable|         40|         27|       1890|
  3|   3|     Port Unreachable|        103|         63|      59335|
  3|   4|  Frag Needed DF Set|          1|         65|       6774|
  3|  10|Dest Host Admin Proh|          4|          5|        162|
  8|   0|         Echo request|         86|        862|     120618|
  8|   1|              UNKNOWN|         18|         36|         96|
  8|   9|              UNKNOWN|          1|        768|       3089|
 11|   0|         TTL exceeded|          4|          5|         26|
 13|   0|   Timestamp request|         18|         36|         96|
 14|   0|     Timestamp Reply|         26|         18|         69|
 15|   0| Information Request|         18|         36|         96|
 17|   0|Address Mask Request|         18|         36|         96|
```

Example 7.10: Exploring Unique ICMP Types/Codes in a SiLK Raw File

- `Echo Request` and `Echo Reply` are used for network host enumeration.
- `Net Unreachable`, `Host Unreachable`, `Port Unreachable`, and `Protocol Unreachable` messages are common responses to port and protocol enumeration.
- `Frag Needed DF Set` messages can result from operating system fingerprinting.

Many of the ICMP types and codes in Example 7.10 serve legitimate purposes. However, Beth knows that the likelihood of deprecated type requests without responses and small counts of source IP addresses "sweeping" the network with anomalous types and codes is low. The results of the `rwuniq` command contain one distinct source IP address sending ICMP echo requests with code 9 to 768 destination IP addresses.

7.3.4 Exploring Attributes of ICMP Messaging

One approach that Beth can take to exploring this data is to analyze common and uncommon attributes of ICMP messaging. On further review of Example 7.10, she identifies three logical groups of common and uncommon ICMP behavior:

- deprecated ICMP types
- uncommon ICMP types
- uncommon behavior of common ICMP types

These logical groupings are useful for understanding information exposure. Beth will now examine them as part of her exploratory analysis.

Exploring Deprecated ICMP Types

Deprecated ICMP types no longer have official support from standards committees and, in many cases, are discouraged due to previous use. The IETF issues both individual and bulk deprecation notices for standards

that the body has introduced. For example, RFC 6633[23] deprecates ICMP source quench message types; RFC 6918[24] deprecates multiple message types.

Searching for these message types in ICMP traffic will help Beth to identify anomalous traffic. Any deprecated types that are seen in monitored traffic should be validated and cataloged for future analysis. Internet research[25] supports this analysis approach.

Beth looks for deprecated ICMP types in the results from Example 7.10. Deprecated type requests do appear in her results. However, there are no corresponding responses. This indicates that information exposure from deprecated types didn't occur on the network during the time period under analysis.

Beth can therefore rule out deprecated ICMP types as a source of information exposure. However, if she did perform a follow-on exploratory analysis, she could identify the sources of these deprecated ICMP type requests. She could also explore how security operations could automate this analysis and display events on a situational awareness dashboard.

Exploring Uncommon ICMP Types

Uncommon ICMP types continue to have official standards committee support, but do not commonly occur in network traffic. Beth notices in the Example 7.10 data that timestamp request and reply traffic occurred during the exploratory analysis period.

ICMP timestamp types are uncommon on most networks[26] and can be used for reconnaissance[27]. Beth therefore decides to find out how many hosts are sending ICMP timestamp reply messages. This will help her to assess the risk of information exposure.

Example 7.11 shows Beth's analytic for counting the number of unique source IP addresses sending timestamp reply messages.

1. The `rwfilter` call in Command 1 partitions ICMP type 14 messages from the exploratory analysis data (`--icmp-type=14`) and passes these records to a file, `icmp_14_all.rw`. This file contains all records that include ICMP timestamp reply messages.

2. The `rwset` call in Command 2 builds an IPset file in memory from the file in Command 1. The `rwsetcat` call counts the number of distinct source IP addresses (26).

```
<1>$ rwfilter icmpall.rw --type=all --icmp-type=14 \
   --pass=icmp_14_all.rw
<2>$ rwset --sip-file=stdout icmp_14_all.rw \
| rwsetcat stdin --count
26
```

Example 7.11: Counting Hosts that Send ICMP Timestamp Reply Messages

[23] Internet Engineering Task Force (IETF) Request for Comments 6633 (https://www.ietf.org/rfc/rfc6633.txt)
[24] Internet Engineering Task Force (IETF) Request for Comments 6918 (https://www.ietf.org/rfc/rfc6918.txt)
[25] Common Attack Pattern Enumeration and Classification 296 (https://capec.mitre.org/data/definitions/296.html)
[26] Cisco Security ICMP Timestamp Request Signature (https://tools.cisco.com/security/center/viewIpsSignature.x?signatureId=2007&signatureSubId=0)
[27] Common Attack Pattern Enumeration and Classification 295 (https://capec.mitre.org/data/definitions/295.html)

CAPEC-295 lists a low severity for timestamp messages. Beth wants to determine if such messages are exiting the subnetwork address space. Example 7.12 shows her analytic for counting the number of source IP addresses that are sending timestamp message replies outside the local subnetworks.

1. The `rwfilter` call in Command 1 partitions the outbound ICMP timestamp messages (`--type=out`) in the file `icmp_14_all.rw` (created in Example 7.11).
2. The output is piped to the `rwset` command to build an IPset that contains the source IP addresses for these outbound messages.
3. Finally, the `rwsetcat` command counts the number of source IP addresses in the set.

The resulting count of zero source IP addresses means that all timestamp replies are contained within the local subnetworks. This rules out ICMP timestamp replies as potential sources of information exposure outside the organization.

```
<1>$ rwfilter icmp_14_all.rw --type=out --pass=stdout \
| rwset --sip-file=stdout \
| rwsetcat stdin --count
0
```

Example 7.12: Counting Hosts that Send ICMP Timestamp Reply Messages Outside their Subnetwork

Beth still would like to check ICMP timestamp messaging on the subnetwork level. Example 7.13 shows how to use SiLK tools to verify the subnetworks that contain these message types.

1. The `rwfilter` call in Command 1 sends all ICMP traffic in the file `icmp_14_all.rw` to standard output.
2. The `rwnetmask` command masks the last 8 bits of the source and destination IP addresses. This provides a subnetwork-level view into the data.
3. Finally, the `rwuniq` command counts the number of subnetworks that sent ICMP timestamp replies.

Beth did uncover several subnetworks that sent ICMP timestamp replies. If she decides to perform follow-on analyses, she can verify whether these message types should occur within the identified subnetworks.

Exploring Uncommon Behavior of Common ICMP Types

Common ICMP types observed in network traffic are echo request (type `8`), echo reply (type `0`), and destination unreachable (type `3`). Uncommon behavior of these types could be a sign of information exposure and Beth decides that it is worth investigating. For example, it would be common for gateway devices or hosts providing network services to respond to echo requests. However, it would be *uncommon* to see these messages from hosts within subnetwork address ranges. If these types of security controls are not enforced, her organization might be vulnerable to information exposure.

Beth's exploratory analysis delves into the uncommon use of common ICMP types and seeks to answer the following questions.

```
<1>$ rwfilter icmp_14_all.rw --type=all --pass=stdout \
| rwnetmask --sip-prefix-length=24 --dip-prefix-length=24 \
| rwuniq --fields=sip,dip --flows
            sIP|            dIP|   Records|
  192.168.142.0|  192.168.142.0|        24|
  192.168.165.0|  192.168.165.0|         6|
   192.168.40.0|   192.168.40.0|         7|
  192.168.164.0|  192.168.164.0|        11|
  192.168.143.0|  192.168.143.0|        15|
  192.168.141.0|  192.168.141.0|         6|
```

Example 7.13: Identifying Networks that Send ICMP Timestamp Reply Messages

Are internal hosts sending echo reply messages to external hosts? Beth first wants to find out if internal hosts are sending echo reply messages to external IP addresses. Example 7.14 shows how she identifies these hosts.

1. The `rwfilter` call in Command 1 partitions the `icmpall.rw` file, which contains all ICMP traffic. It filters outbound echo reply traffic (`--type=out --icmp-type=0`), and uses `internal.set`, the set of IP addresses in internal networks (see Example 7.6) to select both source IP addresses (`--sipset=internal.set`) and IP addresses that aren't destination addresses (`--not-dipset=internal.set`). The records that pass this filter are saved in `icmp_00_external.rw`.

2. The `rwset` call in Command 2 counts the number of receiving destination IP addresses. First, it creates a set of the IP addresses in `icmp_00_external.rw` that are destination addresses.

3. It then pipes the results to `rwsetcat`, which counts the number of IP addresses in the set.

Beth's analysis finds that one IP address receives ICMP echo reply messages. She needs to find out if it is exposing information outside the enterprise network. This could result from improper access control lists or other configuration issues.

```
<1>$ rwfilter icmpall.rw --type=out --icmp-type=0 \
   --sipset=internal.set --not-dipset=internal.set \
   --pass=icmp_00_external.rw
<2>$ rwset --dip-file=stdout icmp_00_external.rw \
| rwsetcat --count stdin
1
```

Example 7.14: Counting External Networks that Receive ICMP Echo Reply Messages

Example 7.15 shows how Beth finds the IP addresses of hosts that are sending and receiving echo reply messages from Example 7.14. The `rwuniq` call in Command 1 counts the hosts in `icmp_00_external.rw` to find out which ones serve as sources and destinations for echo reply messages. Source IP address `10.0.40.21` resides in a service subnetwork, while destination IP address `155.6.3.11` resides in the `Divnet3` subnetwork.

Beth considers this behavior to be normal because the `Divnet3` subnetwork is internal to a theater gateway. However, she still needs to perform a follow-on analysis to verify this trust relationship.

```
<1>$ rwuniq --fields=sip,dip --flows icmp_00_external.rw
            sIP|            dIP|   Records|
     10.0.40.21|     155.6.3.11|        66|
```

Example 7.15: Identifying External Networks that Receive ICMP Echo Reply Messages

Are internal non-gateway hosts sending echo reply messages outside their subnetworks? Beth now turns to the next question in this phase of her exploratory analysis. She wants to identify internal non-gateway hosts that are sending echo reply messages outside their subnetworks. Example 7.16 shows her analytic.

1. The `rwfilter` call in Command 1 partitions `icmpall.rw` to filter all internal source IP addresses that send outbound echo reply messages (`--type=out --icmp-type=0`), using `internal.set`, the set of IP addresses in internal networks (see Example 7.6) to select both. It removes gateway and service hosts as sources and saves the results to the `icmp_00_internal.rw` raw file.

2. The `rwset` and `rwsetcat` calls in Command 2 count the number of source IP addresses that exhibit this behavior.

Beth examines the results in Example 7.16 and finds 20 non-gateway source IP addresses that send echo reply messages outside their subnetworks.

```
<1>$ rwfilter icmpall.rw --type=out --icmp-type=0 \
   --sipset=internal.set --pass=stdout \
| rwfilter stdin --not-sipset=gateways.set --pass=stdout \
| rwfilter stdin --not-sipset=services.set \
   --pass=icmp_00_internal.rw
<2>$ rwset --sip-file=stdout icmp_00_internal.rw \
| rwsetcat --count stdin
20
```

Example 7.16: Counting Internal Non-Gateway Hosts that Send ICMP Echo Reply Messages Outside their Subnetwork

Before reviewing the specific hosts identified in Example 7.16 further, Beth decides to label the results to better understand internal service usage for the different subnetworks. Example 7.17 shows how she builds an IPv4 prefix map that provides labels for various internal network services. Her pmap labels the subnetworks that host the `DMZ`, `SERVICES`, and `VPNPOOL` services.

In Example 7.18, Beth uses this pmap with the `rwuniq` command to analyze host behavior. Echo reply messages from source IP address `192.168.70.10` (proxy device) to destination IP address `10.0.40.27` (Nagios server) are likely to be legitimate.

However, the remaining results should be considered suspicious. The multiple echo reply messages from an internal subnetwork host to a `VPNPOOL` host (`192.168.181.8`) and a domain controller (`10.0.40.20`) are not considered to be common ICMP echo reply behavior.

Beth should investigate this behavior further in follow-on analyses to find out if it is malicious.

```
<1>$ cat <<-EOF >servicenets.pmap.txt
# Service Network Descriptions
map-name services
mode ipv4

10.0.20.0/24     DMZ      # DMZ subnet
10.0.40.0/24     SERVICES # Services subnet
192.168.20.0/24  DMZ      # DMZ subnet
192.168.40.0/24  SERVICES # Services subnet
192.168.181.0/24 VPNPOOL  # VPN pool
EOF
<2>$ rwpmapbuild --input-file=servicenets.pmap.txt \
   --output-file=servicenets.pmap
<3>$ file servicenets.pmap.txt servicenets.pmap
servicenets.pmap.txt: ASCII text
servicenets.pmap:     SiLK, PREFIXMAP v2, Little Endian, Uncompressed
```

Example 7.17: Building a Prefix Map of Service Subnetworks

```
<1>$ rwuniq --pmap-file=services:servicenets.pmap \
   --fields=sip,dip,itype,icode,dst-services --flows \
   icmp_00_internal.rw
            sIP|            dIP|iTy|iCo|dst-services|   Records|
 192.168.121.77|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.166.55|  192.168.181.8|  0|  0|      VPNPOOL|         5|
  192.168.121.2|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.166.12|  192.168.181.8|  0|  0|      VPNPOOL|         5|
  192.168.111.2|  192.168.181.8|  0|  0|      VPNPOOL|         5|
  192.168.121.2|  192.168.181.8|  0|  9|      VPNPOOL|         5|
192.168.111.223|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.111.212|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.166.15|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.111.109|  192.168.181.8|  0|  0|      VPNPOOL|         5|
  192.168.70.10|     10.0.40.27|  0|  0|    SERVICES|      1013|
192.168.121.158|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.124.218|  192.168.40.20|  0|  0|    SERVICES|         2|
 192.168.166.43|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.121.57|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.111.131|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.166.43|  192.168.40.20|  0|  0|    SERVICES|         2|
  192.168.111.2|  192.168.181.8|  0|  9|      VPNPOOL|         5|
192.168.121.145|  192.168.181.8|  0|  0|      VPNPOOL|         5|
 192.168.111.94|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.166.233|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.121.244|  192.168.181.8|  0|  0|      VPNPOOL|         5|
192.168.124.205|     10.0.40.20|  0|  0|    SERVICES|         4|
```

Example 7.18: Identifying Internal Non-Gateway Hosts that Send ICMP Echo Reply Messages Outside their Subnetwork

7.3.5 Wrap Up Investigation and Identify Follow-On Analyses

In this case study, Beth used network flow data and SiLK tools to perform an exploratory analysis of ICMP traffic to assess the risk of information exposure. Her analysis shows that information exposure from ICMP messaging does not appear to be widespread across the enterprise.

However, Beth did find two possible trouble spots: trust relationships between internal subnetworks and suspicious activity associated with ICMP echo reply messages. She should investigate them further to determine if they represent scanning or other malicious activity. Her follow-on analyses may require support and data from outside the security operations group.

This page intentionally left blank.

Chapter 8

Extending the Reach of SiLK with PySiLK

This chapter discusses how to use PySiLK, the SiLK Python extension, to support analyses that are difficult to implement within the normal constraints of the SiLK tool suite. Sometimes, an analyst needs to use parts of the SiLK suite's native functionality in a modified way. The capabilities of PySiLK simplify these analyses.

This chapter does not discuss the issues involved in composing new PySiLK scripts or how to code in the Python programming language. Several example scripts are shown, but the detailed design of each script will not be presented here.

Upon completion of this chapter, you will be able to

- explain the purpose of PySiLK in analysis
- use and modify PySiLK plug-ins to match records in `rwfilter`
- use and modify PySiLK plug-ins to add fields for `rwcut` and `rwsort`
- use and modify PySiLK plug-ins to add key fields and summary values for `rwuniq` and `rwstats`

Additional PySiLK and Python programming language resources include

- a brief guide to coding PySiLK plug-ins, provided by the `silkpython` manual page (see `man silkpython` or Section 3 of *The SiLK Reference Guide* at https://tools.netsa.cert.org/silk/reference-guide.pdf)
- detailed descriptions of the PySiLK structures, provided in *PySiLK: SiLK in Python* (https://tools.netsa.cert.org/silk/pysilk.pdf)
- generic programming in the Python programming language, as described in many locations on the web, particularly on the Python official website (https://www.python.org/doc/)

8.1 Using PySiLK

PySiLK is an extension to the SiLK tool suite that expands its functionality via scripts written in the Python programming language. The purpose of PySiLK is to support analytical use cases that are difficult to express, implement, and support with the capabilities natively built into SiLK, while using those capabilities where appropriate.

To extend the SiLK tools with PySiLK, first write a Python file that calls Python functions defined in the `silk.plugin` Python module. To use this Python file, specify the `--python-file` switch for one of the SiLK tools that supports PySiLK. (The `rwfilter`, `rwstats`, `rwuniq`, `rwcut`, and `rwsort` tools can all make use of PySiLK extensions.) The tool then loads the Python file and makes the new functionality available.

8.1.1 PySiLK Requirements

To use PySiLK:

1. Install the appropriate version of the Python language[28] on your system.
2. Load the PySiLK library (a directory named `silk`) in the `site-packages` directory of the Python installation.
3. To ensure that the PySiLK library works properly, set the `PYTHONPATH` environment variable to include the `site-packages` directory.

8.1.2 PySiLK Scripts and Plug-ins

PySiLK code comes in two forms: standalone Python programs and plug-ins for SiLK tools. Both of these forms use a Python module named `silk` that is provided as part of the PySiLK library. PySiLK provides the capability to manipulate SiLK objects (flow records, IPsets, bags, etc.) with Python code.

For analyses that will not be repeated often or that are expected to be modified frequently, the relative brevity of PySiLK renders it an efficient alternative. As with all programming, analysts need to use algorithms that meet the speed and space constraints of the project. Often (but not always), building upon the processing of the SiLK tools by use of a plug-in yields a more suitable solution than developing a stand-alone script.

PySiLK plug-ins for SiLK tools use an additional component called `silkpython`, which is also provided with the PySiLK library. The `silkpython` component creates the application programming interfaces (APIs), simple and advanced, that connect a plug-in to SiLK tools. Currently, `silkpython` supports the following SiLK tools: `rwfilter`, `rwcut`, `rwgroup`, `rwsort`, `rwstats`, and `rwuniq`.

- For `rwfilter`, `silkpython` permits a plug-in to provide new types of partitioning criteria.
- For `rwcut`, plug-ins can create new flow record fields for display.
- For `rwgroup` and `rwsort`, plug-ins can create fields to be used as all or part of the key for grouping or sorting records.

[28] Python 2.7.x for SiLK Version 3.19.1.

- For `rwstats` and `rwuniq`, two types of fields can be defined: *key* fields used to categorize flow records into bins and *summary value* fields used to compute a single value for each bin from the records in those bins.

- For all of the tools, `silkpython` allows a plug-in to create new SiLK tool switches (parameters) to modify the behavior of the aforementioned partitioning criteria and fields.

The `silkpython` module provides only one function for establishing (registering) partitioning criteria (filters) for `rwfilter`. The simple API provides four functions for creating key fields for the other five supported SiLK tools and three functions for creating summary value fields for `rwstats` and `rwuniq`. The advanced API provides one function that can create either key fields or summary value fields, and permits a higher degree of control over these fields.

8.2 Extending `rwfilter` with PySiLK

PySiLK extends the capabilities of `rwfilter` by letting the analyst create new methods for partitioning flow records.[29]

For a single execution of `rwfilter`, PySiLK is much slower than using a combination of `rwfilter` parameters and usually slower than using a C-language plug-in. However, there are several ways in which using PySiLK can replace a series of several `rwfilter` executions with a single execution and ultimately speed up the overall process.

Without PySiLK, `rwfilter` has limitations using its built-in partitioning parameters:

- Each flow record is examined without regard to other flow records. That is, no state is retained.

- There is a fixed rule for combining partitioning parameters: Any of the alternative values within a parameter satisfies that criterion (i.e., the alternatives are joined implicitly with a logical *or* operation). All parameters must be satisfied for the record to pass (i.e., the parameters are joined implicitly with a logical *and* operation).

- The types of external data that can assist in partitioning records are limited. IP sets, tuple files, and prefix maps are the only types provided by built-in partitioning parameters.

PySiLK is useful to expand the capabilities of `rwfilter` in these cases:

- Information from prior records may help to partition subsequent records into the pass or fail categories.

- A series of nontrivial alternatives form the partitioning condition.

- The partitioning condition employs a control structure or data structure.

[29] These partitioning methods may also calculate values not normally part of `rwfilter`'s output, typically emitting those values to a file or to the standard error stream to avoid conflicts with flow record output.

8.2.1 Using PySiLK to Incorporate State from Previous Records: Eliminating Inconsistent Sources

For an example of where some information (or *state*) from prior records may help in partitioning subsequent records, consider Example 8.1. This script (`ThreeOrMore.py`) passes all records that have a source IP address used in two or more prior records. This can be useful if you want to eliminate casual or inconsistent sources of particular behavior. The `addrRefs` variable is the record of how many times each source IP address has been seen in prior records. The `threeOrMore` function holds the Python code to partition the records. If it determines the record should be passed, it returns `True`; otherwise it returns `False`.

```
import sys   # stderr

bound = 3    # default threshold for passing record
addrRefs={}  # key = IP address, value = reference count

def threeOrMore(rec):
  global addrRefs  # allow modification of addrRefs

  keyval = rec.sip  # change this to count on different field
  addrRefs[keyval] = addrRefs.get(keyval, 0) + 1
  return addrRefs[keyval] >= bound

def set_bound(integer_string):
  global bound

  try:
    bound = int(integer_string)
  except ValueError:
    print >>sys.stderr, '--limit value, %s, is not an integer.' % integer_string

def output_stats():
  AddrsWithEnufFlows = len([1 for k in addrRefs.keys()
                            if addrRefs[k] >= bound])
  print >>sys.stderr, 'SIPs: %d;  SIPs meeting threshold: %d' % (len(addrRefs),
                                                                 AddrsWithEnufFlows)

register_filter(threeOrMore, finalize=output_stats)
register_switch('limit', handler=set_bound, help='Threshold for passing')
```

Example 8.1: `ThreeOrMore.py`: Using PySiLK for Memory in `rwfilter` Partitioning

In Example 8.1, the call to `register_filter` informs `rwfilter` (through `silkpython`) to invoke the specified Python function (`threeOrMore`) for each flow record that has passed all the built-in SiLK partitioning criteria. In the `threeOrMore` function, the `addrRefs` dictionary is a container that holds entries indexed by an IP address and whose values are integers.

When the `get` method is applied to the dictionary, it obtains the value for the entry with the specified

key, `keyval`, if such an entry already exists. If this is the first time that a particular key value arises, the `get` method returns the supplied default value, zero. Either way, one is added to the value obtained by `get`. The `return` statement compares this incremented value to the `bound` threshold value and returns the Boolean result to `silkpython`, which informs `rwfilter` whether the current flow record passes the partitioning criterion in the PySiLK plug-in.

In Example 8.1, the `set_bound` function is not required for the partitioning to operate. It provides the capability to modify the threshold that the `threeOrMore` function uses to determine which flow records pass. The call to `register_switch` informs `rwfilter` (through `silkpython`) that the `--limit` parameter is acceptable in the `rwfilter` command after the `--python-file=ThreeOrMore.py` parameter, and that if the user specifies the parameter (e.g., `--limit=5`) the `set_bound` function will be run to modify the `bound` variable before any flow records are processed. The value provided in the `--limit` parameter will be passed to the `set_bound` function as a string that needs to be converted to an integer so it can participate later in numerical comparisons.

If the user specifies a string that is not a representation of an integer, the conversion will fail inside the `try` statement, raising a `ValueError` exception and displaying an error message; in this case, `bound` is not modified.

8.2.2 Using PySiLK to Incorporate State from Previous Records: Detecting Port Knocking

For an example in which some information (or *state*) from prior records may help in partitioning subsequent records, consider *port knocking*. This is a technique used to thwart port scanning. Port scanning involves sending single packets to particular ports on a target host to see what response, if any, is returned by the target. The response, or lack of one, is interpreted to determine if there is a service available on that port on the target host. Port knocking is employed by the administrator of the target host to make all ports look as if there are no services available on any of them.

Port knocking requires legitimate users (or their software) to know the secret combination of actions that must be taken before an attempt is made to connect to a service port. These actions consist of attempts to connect (or *knocks*) to certain other ports in the correct order right before attempting a connection to the service port. Achieving the correct sequence of knocks creates a temporary rule in the firewall to allow the sender of the knocks to connect to a particular service. Profiling a network to obtain situational awareness could include port knocking detection to explain what would otherwise look like strange traffic.

Example 8.2 implements a plug-in for `rwfilter` that takes a simple approach. The plug-in requires that its input be sorted by IP addresses and time. That way the port knocks appear in the input right before the service connection attempt. This means that the plug-in needs to retain state for only one connection attempt at a time, simplifying the code and greatly reducing the memory requirements.

The plug-in looks for three consecutive flow records where the first two (the port knocks) attempt to initiate TCP connections to different ports, but there are no following packets with the ACK flag that would indicate the connection had been established. After the two port knock flows, there must be a flow for a third port which does have additional packets with the ACK flag.

These criteria are somewhat simple. We could add constraints such as specifying that the three ports

must have a certain ordinal relationship (e.g., lower-higher-lower). However, Example 8.2 shows the essential elements. When the three-flow sequence is found, the third flow passes the filter and a text record is displayed.

```
import datetime # timedelta()
import sys # stdout

REQDKNOCKS = 2  # number of required knocks with distinct port numbers
INTERVAL = datetime.timedelta(seconds=5)  # knocks & conn this close in time
TCP = 6  # protocol number

portListWidth = REQDKNOCKS * 7
lastsip = None

def note_first_knock(rec):
   global lastsip, lastdip, portlist, lastetime
   lastsip = rec.sip
   lastdip = rec.dip
   portlist = [rec.dport]
   lastetime = rec.etime
   return

def examine_flow(rec):
   global lastsip, lastdip, portlist, lastetime
   if (rec.protocol == TCP and rec.initial_tcpflags is not None and
       rec.initial_tcpflags.matches('S/SA')):  # initial SYN (client to server)
      if lastsip is not None and rec.sip == lastsip and rec.dip == lastdip:
         if rec.stime - lastetime <= INTERVAL:
            if rec.session_tcpflags.ack:  # established connection
               # connected to knocked port or insufficient knocks?
               if rec.dport in portlist or len(portlist) < REQDKNOCKS:
                  lastsip = None
               else:  # enough prior knocks?
                  sys.stdout.write('%15s %15s %*s %5d\n' % (lastsip, lastdip,
                                       portListWidth, portlist, rec.dport))
                  lastsip = None
                  return True  # flow record passes filter
            else:  # connection not established; just a knock
               if rec.dport in portlist:  # already seen this port
                  note_first_knock(rec)  # start over as 1st knock
               else:
                  if len(portlist) >= REQDKNOCKS:  # add a knock
                     del portlist[0]  # delete oldest knock, make room for new
                  portlist.append(rec.dport)
                  lastetime = rec.etime
         # last knock was too long ago
         elif rec.session_tcpflags.ack:  # established connection
            lastsip = None
         else:  # too long ago and connection not established; just a knock
            note_first_knock(rec)  # start over as 1st knock
      # new sip and/or dip
      elif not rec.session_tcpflags.ack: # conn not established; just a knock
         note_first_knock(rec)  # start over as 1st knock
   return False  # flow record fails filter
```

```
def show_heading():
  sys.stdout.write('%15s %15s %*s %5s\n' % ('sIP', 'dIP', portListWidth,
                                            'Knock-Ports', 'Estab'))

register_filter(examine_flow, initialize=show_heading)
```

Example 8.2: `portknock.py`: Using PySiLK to Retain State in `rwfilter` Partitioning

8.2.3 Using PySiLK with `rwfilter` in a Distributed or Multiprocessing Environment

An analyst could use a PySiLK script with `rwfilter` by first calling `rwfilter` to retrieve the records that satisfy a given set of conditions, then piping those records to a second `rwfilter` call that uses the `--python-file` parameter to invoke the script. This is shown in Example 8.3. This syntax is preferred to simply including the `--python-file` parameter on the first call, since its behavior is more consistent across execution environments. If `rwfilter` is running on a multiprocessor configuration, running the script on the first `rwfilter` call cannot be guaranteed to behave consistently for a variety of reasons, so running PySiLK scripts via a piped `rwfilter` call is more consistent.

```
<1>$ rwfilter --start-date=2015/06/02 --end-date=2015/06/18 \
   --type=inweb --protocol=6 --dport=443 \
   --bytes-per-packet=65- --packets=4- \
   --flags-all=SAF/SAF,SAR/SAR --pass=stdout \
| rwfilter stdin --python-file=ThreeOrMore.py \
   --pass=web.rw
SIPs: 81;  SIPs meeting threshold: 81
```

Example 8.3: Calling `ThreeOrMore.py`

8.2.4 Simple PySiLK with `rwfilter --python-expr`

Some analyses that do not lend themselves to solutions with just the SiLK built-in partitioning parameters may be so simple with PySiLK that they center on an expression that evaluates to a Boolean value. Using the `rwfilter --python-expr` parameter will cause `silkpython` to provide the rest of the Python plug-in program.

Example 8.4 partitions flow records that have the same port number for their source port and destination port (`sport` and `dport`). Although the name for the flow record object is specified by a function parameter in user-written Python files, with `--python-expr`, the record object is always called `rec`. The source port and destination port therefore can be specified as `rec.sport` and `rec.dport`. Checking whether their values are equal becomes very simple.

With `--python-expr`, it is not possible to retain state from previous flow records as in Example 8.1. Nor is it possible to incorporate information from sources other than the flow records. Both of these require a

```
<1>$ rwfilter flows.rw --protocol=6,17 --python-expr='rec.sport==rec.dport' \
              --pass=equalports.rw
```

Example 8.4: Using `--python-expr` for Partitioning

plug-in invoked by `--python-file`.

8.2.5 PySiLK with Complex Combinations of Rules

Example 8.5 shows an example of using PySiLK to filter for a condition with several alternatives. This code is designed to identify virtual private network (VPN) traffic in the data, using IPsec, OpenVPN®, or VPNz®. This involves having several alternatives, each matching traffic either for a particular protocol (50 or 51) or for particular combinations of a protocol (17) and ports (500, 1194, or 1224). This could be done using a pair of `rwfilter` calls (one for UDP [17] and one for both ESP [50] and AH [51]) and `rwcat` to put them together, but this is less efficient than using PySiLK.

```
def vpnfilter(rec):
  return ((rec.protocol == 17 and  # UDP
          (rec.dport in (500, 1194, 1224) or   # IKE, OpenVPN, VPNz
           rec.sport in (500, 1194, 1224)   ) )
       or rec.protocol in (50, 51) )  # ESP, AH

register_filter(vpnfilter)
```

Example 8.5: `vpn.py`: Using PySiLK with `rwfilter` for Partitioning Alternatives

8.2.6 Use of Data Structures in Partitioning

Example 8.6 shows the use of a data structure in an `rwfilter` condition. This particular case identifies internal IP addresses responding to contacts by IP addresses in certain external blocks. The difficulty is that the response is unlikely to go back to the contacting address and likely instead to go to another address on the same network. Matching this with conventional `rwfilter` parameters is very slow and repetitive. By building a list of internal IP addresses and the networks they've been contacted by, `rwfilter` can partition records based on this list using the PySiLK script in Example 8.6, called `matchblock.py`.

In Example 8.6, lines 1 and 2 import objects from two modules. Line 3 sets a constant (with a name in all uppercase by convention). Line 4 creates a global variable to hold the name of the file containing external

netblocks and gives it a default value. Lines 6, 10, and 32 define functions to be invoked later. Line 42 informs `silkpython` of two things: (1) that the `open_blockfile` function should be invoked after all command-line switches (parameters) have been processed and before any flow records are read and (2) that in addition to any other partitioning criteria, every flow record must be tested with the `match_block` function to determine if it passes or fails. Line 43 tells `silkpython` that `rwfilter` should accept a `--blockfile` parameter on the command line and process its value with the `change_blockfile` function before the initialization function, `open_blockfile`, is invoked.

When `open_blockfile` is run, it builds a list of external netblocks for each specified internal address. Line 25 converts the specified address to a PySiLK address object; if that's not possible, a `ValueError` exception is raised, and that line in the blockfile is skipped. Line 26 similarly converts the external netblock specification to a PySiLK IP wildcard object; if that's not possible, a `ValueError` exception is raised, and that line in the file is skipped. Line 26 also appends the netblock to the internal address's list of netblocks; if that list does not exist, the `setdefault` method creates it.

When each flow record is read by `rwfilter`, `silkpython` invokes `match_block`, which tests every external netblock in the internal address's list to see if it contains the external, destination address from the flow record. If an external address is matched to a netblock in line 35, the test passes. If no netblocks in the list match, the test fails in line 39. If there is no list of netblocks for an internal address (because it was not specified in the blockfile), the test fails in line 38.

Example 8.7 uses command-line parameters to invoke the Python plug-in and pass information to the plug-in script (specifically the name of the file holding the block map). Command 1 displays the contents of the block map file. Each line has two fields separated by a comma. The first field contains an internal IP address; the second field contains a wildcard expression (which could be a CIDR block or just a single address) describing an external netblock that has communicated with the internal address. Command 2 then invokes the script using the syntax introduced previously, augmented by the new parameter.

```
from silk import IPAddr,IPWildcard
import sys  # exit(), stderr
PLUGIN_NAME = 'matchblock.py'
blockname='blocks.csv'

def change_blockfile(block_str):
  global blockname
  blockname = block_str

def open_blockfile():
  global blockfile, blockdict
  try:
    blockfile = open(blockname)
  except IOError, e_value:
     sys.exit('%s: Block file: %s' % (PLUGIN_NAME, e_value))
  blockdict = dict()
  for line in blockfile:
    if line.lstrip()[0] == '#':  # recognize comment lines
      continue                   # skip entry
    fields = line.strip().split(',') # remove NL and split fields on commas
    if len(fields) < 2:  # too few fields?
      print >>sys.stderr, '%s: Too few fields: %s' % (PLUGIN_NAME, line)
      continue           # skip entry
    try:
      idx = IPAddr(fields[0].rstrip())
      blockdict.setdefault(idx, []).append(IPWildcard(fields[1].strip()))
    except ValueError:   # field cannot convert to IPAddr or IPWildcard
      print >>sys.stderr, '%s: Bad address or wildcard: %s' % (PLUGIN_NAME, line)
      continue           # skip entry
  blockfile.close()

def match_block(rec):
  try:
    for netblock in blockdict[rec.sip]:
      if rec.dip in netblock:
        return True
  except KeyError:  # no such inside addr
    return False
  return False  # no netblocks match

#  M A I N
register_filter(match_block, initialize=open_blockfile)
register_switch('blockfile', handler=change_blockfile,
  help='Name of file that holds CSV block map.  Def. blocks.csv')
```

Example 8.6: `matchblock.py`: Using PySiLK with `rwfilter` for Structured Conditions

```
<1>$ cat blockfile.csv
198.51.100.17, 192.168.0.0/16
203.0.113.178, 192.168.x.x
<2>$ rwfilter out_month.rw --protocol=6 --dport=25 --pass=stdout \
      | rwfilter stdin --python-file=matchblock.py \
                  --blockfile=blockfile.csv --print-statistics
Files     1.  Read     375567.  Pass          8. Fail      375559.
```

Example 8.7: Calling `matchblock.py`

8.3 Extending SiLK with Fields Defined with PySiLK

Five SiLK tools support fields defined with PySiLK: `rwcut`, `rwgroup`, `rwsort`, `rwstats`, and `rwuniq`. All five support newly defined *key* fields, although for `rwcut` these fields are not really the key to any sorting or grouping operation; they are simply available for display. Two of the tools, `rwstats` and `rwuniq`, support newly defined *summary value* fields. For fields that require the use of the advanced API, the programmer will want to determine which of the five tools will be used with these fields to avoid unnecessary programming. By examining the characteristics of the new field the programmer can determine which tools can take advantage of the field.

`rwcut` can make use of any key field that produces character strings (text). It does not matter if field values can be used in computations, comparisons, or sorting. As long as the field can be represented as text, `rwcut` can use it. When registering such fields, a function to produce a text value must be provided.

For `rwgroup` and `rwsort` to utilize a field defined with PySiLK, the field registration must provide a function that produces binary (non-text) values that can be compared. For `rwgroup`, the comparison is only for equality to determine if two consecutive records belong in the same group. For `rwsort` the comparison must also determine the order of unequal values.

For `rwstats` and `rwuniq`, a key field not only needs a binary representation for grouping purposes, it also needs a function to convert the binary key (bin) to text for display purposes. Furthermore, it must make sense semantically for the field to produce a many-to-one mapping from its inputs to its value. This is necessary for the field to make a good key (or partial key) for bins. A field makes a poor binning key if nearly every record produces a unique field value, or if nearly every record produces the same field value.

8.4 Extending `rwcut` and `rwsort` with PySiLK

PySiLK is useful with `rwcut` and `rwsort` in these cases:

- An analysis requires a value based on a combination of fields, possibly from a number of records.
- An analyst chooses to use a function on one or more fields, possibly conditioned by the value of one or more fields. The function may incorporate data external to the records (e.g., a table of header lengths).

8.4.1 Computing Values from Multiple Records

Example 8.8 shows the use of PySiLK to calculate a value from the same field of two different records in order to provide a new column to display with `rwcut`. This particular case, which will be referred to as `delta.py`, introduces a `delta_msec` column, with the difference between the start time of two successive records. Potential uses for this column including ready identification of flows that occur at very stable intervals, such as keep-alive traffic or beaconing.

The plug-in uses global variables to save the IP addresses and start time between records and then returns to `rwcut` the number of milliseconds between start times. The `register_int_field` call allows the use of `delta_msec` as a new field name and gives `rwcut` the information that it needs to process the new field.

```
last_sip = None

def compute_delta(rec):
    global last_sip, last_dip, last_time
    if last_sip is None or rec.sip != last_sip or rec.dip != last_dip:
        last_sip = rec.sip
        last_dip = rec.dip
        last_time = rec.stime_epoch_secs
        deltamsec = 0
    else:  # sip and dip same as previous record
        deltamsec = int(1000. * (rec.stime_epoch_secs - last_time))
        last_time = rec.stime_epoch_secs
    return deltamsec

register_int_field ('delta_msec', compute_delta, 0, 4294967295)
#                    fieldname      function    min     max
```

Example 8.8: `delta.py`

To use `delta.py`, Example 8.9 sorts the flow records by source address, destination address, and start time after pulling them from the repository. After sorting, the example passes them to `rwcut` with the `--python-file=delta.py` parameter before the `--fields` parameter so that the `delta_msec` field name is defined. Because of the way the records are sorted, if the source or destination IP addresses are different in two consecutive records, the latter record could have an earlier `sTime` than the prior record. Therefore, it makes sense to compute the time difference between two records only when their source addresses match and their destination addresses match. Otherwise, the delta should display as zero.

8.4.2 Computing a Value Based on Multiple Fields in a Record

```
<1>$ rwfilter --type=out --start-date=2015/06/02 \
   --end-date=2015/06/18 --protocol=17 --packets=1 \
   --pass=stdout \
| rwsort --fields=sIP,dIP,sTime \
| rwcut --python-file=delta.py \
   --fields=sIP,dIP,sTime,delta_msec --num-recs=20
           sIP|            dIP|                  sTime|delta_msec|
    10.0.20.58|    128.8.10.90|2015/06/17T20:50:50.725|         0|
    10.0.20.58|    128.8.10.90|2015/06/17T20:50:50.725|         0|
    10.0.20.58|    128.8.10.90|2015/06/17T20:50:50.725|         0|
    10.0.20.58|    199.7.83.42|2015/06/17T20:50:46.717|         0|
    10.0.20.58|    199.7.83.42|2015/06/17T20:50:46.717|         0|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:08.030|         0|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:08.268|       237|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:08.580|       312|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:08.580|         0|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:08.674|        94|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:09.015|       341|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:09.043|        27|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:09.215|       171|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:09.397|       182|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:10.228|       830|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:10.620|       391|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:10.622|         2|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:10.622|         0|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:11.069|       447|
    10.0.40.20|     10.0.20.58|2015/06/16T12:48:11.415|       345|
```

Example 8.9: Calling `delta.py`

Example 8.10 shows the use of a PySiLK plug-in for both `rwsort` and `rwcut` that supplies a value calculated from several fields from a single record. In this example, the new value is the number of bytes of payload conveyed by the flow. The number of bytes of header depends on the version of IP as well as the Transport-layer protocol being used (IPv4 has a 20-byte header, IPv6 has a 40-byte header, and TCP adds 20 additional bytes, while UDP adds only 8 and GRE [protocol 47] only 4, etc.).

The `header_len` variable holds a mapping from protocol number to header length. Protocols omitted from the mapping contribute zero bytes for the Transport-layer header. This is then multiplied by the number of packets and subtracted from the flow's byte total. This code assumes no packet fragmentation is occurring. The same function is used to produce both a value for `rwsort` to compare and a value for `rwcut` to display, as indicated by the `register_int_field` call.

```
#            ICMP  IGMP  IPv4    TCP    UDP    IPv6   RSVP
header_len={1:8,  2:8,  4:20,  6:20,  17:8,  41:40,  46:8,
            47:4,  50:8,  51:12,  88:20,  132:12}
#             GRE    ESP     AH    EIGRP    SCTP

def bin_payload(rec):
    transport_hdr = header_len.get(rec.protocol, 0)
    if rec.is_ipv6():
        ip_hdr = 40
    else:
        ip_hdr = 20
    return rec.bytes - rec.packets * (ip_hdr + transport_hdr)

register_int_field('payload', bin_payload, 0, (1 << 32) - 1)
#                  fieldname    function  min       max
```

Example 8.10: `payload.py`: Using PySiLK for Conditional Fields with `rwsort` and `rwcut`

Example 8.11 shows how to use Example 8.10 with both `rwsort` and `rwcut`. The records are sorted into payload-size order and then output, showing both the bytes and payload values.

8.4.3 Defining a Character String Field for `rwcut`

PySiLK has no function in the simple API for creating character-string fields. Do not be tempted to use an enumeration where there are many possible strings produced for the field; an enumeration can be quite memory-intensive. It will not sort correctly in `rwsort` or in `rwuniq` with the `--sort-output` parameter.

Creating a character-string field with the advanced API (`register_field` function) is not difficult. Starting with an example of a string field that works only with `rwcut`, Example 8.12 is a PySiLK plug-in that makes large values in the built-in `duration` field more understandable by breaking it down into days, hours, minutes, and seconds (including milliseconds). This field does not provide a binary value, so the field has no usefulness with `rwsort` or `rwgroup`, which produce non-text output. Since this field embodies the same information as the built-in `duration` field, the built-in field is better suited for use with these tools as it will yield much better performance. The usefulness of `decode_duration` with `rwstats` and `rwuniq` is

```
<1>$ rwsort inbound.rw --python-file=payload.py --fields=payload \
      | rwcut --python-file=payload.py --fields=5,packets,bytes,payload
pro|   packets|     bytes|   payload|
  6|      1007|     40280|         0|
  1|         1|        28|         0|
  6|         2|        92|        12|
  6|         8|       332|        12|
  1|         1|        48|        20|
 17|         1|        51|        23|
  1|        14|       784|       392|
 80|         3|       762|       702|
 50|        16|      1920|      1472|
 17|         1|      2982|      2954|
 47|       153|     12197|      8525|
 50|        77|     11088|      8932|
 17|       681|    212153|    193085|
  6|       309|    398924|    386564|
 51|      5919|    773077|    583669|
 51|     10278|   1344170|   1015274|
  6|       820|   1144925|   1112125|
 97|   2134784|2770949632|2728253952|
```

Example 8.11: Calling `payload.py`

dubious as well; although these tools produce textual output, the lack of a many-to-one mapping makes this field an ideal candidate for use only with `rwcut`.

Example 8.13 shows both the built-in `duration` field and the associated `decode_duration` field for several flow records.

```
'''Define a field for rwcut which formats the flow duration as a string
   representation of days, hours, minutes, seconds, and milliseconds.
'''

SECPERHOUR = 3600
SECPERMIN = 60

def decode_duration(rec):
  (hours, seconds) = divmod(rec.duration.seconds, SECPERHOUR)
  (minutes, seconds) = divmod(seconds, SECPERMIN)
  return '%02dd%02dh%02dm%02d.%03ds' % (rec.duration.days, hours, minutes,
                                        seconds, rec.duration.microseconds // 1000)

register_field('decode_duration', column_width=16, rec_to_text=decode_duration,
               description='Decomposition of duration into days, hours,'
                           ' minutes, seconds, and milliseconds')
```

Example 8.12: `decode_duration.py`: A Program to Create a String Field for `rwcut`

```
<1>$ rwcut flows.rw --python-file=decode_duration.py \
                    --fields=decode_duration,duration
 decode_duration| duration|
01d07h32m49.161s|113569.161|
00d00h29m00.450s| 1740.450|
00d19h05m22.667s|68722.667|
00d00h00m03.000s|    3.000|
```

Example 8.13: Calling `decode_duration.py`

8.4.4 Defining a Character String Field for Five SiLK Tools

A plug-in to create a character-string field not only for `rwcut`, but also for `rwgroup`, `rwsort`, `rwstats`, and `rwuniq` needs a little more code. In Example 8.14 a regular expression is used to provide a pattern for the site-name portion of a sensor name. Since the regular expression does not permit numeric digits as part of the site name, the pattern matching ends when a digit is reached in the sensor name. In addition to defining a function that derives a string from a SiLK flow record, we need a function to pad the strings so their lengths are the same for all records, and a function to strip the padding.

The functions provided here should work for other string-field plug-ins, except for the call to `get_site`, which derives the string from the record. Then we must establish the maximum length of the field, and supply a few additional parameters to the `register_field` call. Example 8.15 shows how use of the derived field reduces the output to fewer lines than the number of sensors.

```
import re  # compile(), SRE_Pattern.match(), SRE_Match.group()

# Global variables that may be modified for the enterprise.
MAX_FIELD_LEN = 8
regex = re.compile(r"^[-A-Za-z]+")

def get_site(rec):
  '''Derive site name from sensor field in flow record.'''

  return regex.match(rec.sensor).group()

def remove_padding(bin):
  '''Remove padding from fixed-length string.'''

  # for Python 3.x, change "bin" to "str(bin, 'UTF-8')"
  return bin.rstrip('\0')

def pad_to_fixed_length(rec):
  '''Make site names all the same length.'''

  # for Python 3.x, enclose the entire expression in "bytes( , 'UTF-8')"
  return get_site(rec).ljust(MAX_FIELD_LEN, '\0')

register_field('Site',
               bin_bytes=MAX_FIELD_LEN,
               bin_to_text=remove_padding,
               column_width=MAX_FIELD_LEN,
               description='Site name derived from sensor.',
               rec_to_bin=pad_to_fixed_length,
               rec_to_text=get_site)
```

Example 8.14: `sitefield.py`: A Program to Create a String Field for Five SiLK Tools

```
<1>$ rwcut flows.rw --python-file=sitefield.py --fields=sensor,Site
  sensor|    Site|
NewYork1| NewYork|
NewYork2| NewYork|
 London0|  London|
London1a|  London|
NewYork1| NewYork|
NewYork2| NewYork|
NewYork3| NewYork|
 London0|  London|
London1a|  London|
London1a|  London|
London1a|  London|
NewYork2| NewYork|
NewYork2| NewYork|
NewYork3| NewYork|
NewYork3| NewYork|
NewYork2| NewYork|
London1a|  London|
<2>$ rwuniq flows.rw --python-file=sitefield.py --field=Site --sort-output
    Site|   Records|
  London|         7|
 NewYork|        10|
```

Example 8.15: Calling `sitefield.py`

8.5 Defining Key Fields and Summary Value Fields for `rwuniq` and `rwstats`

In addition to defining key fields that `rwcut` can use for display, `rwsort` can use for sorting, and `rwgroup` can use for grouping, `rwuniq` and `rwstats` can make use of both key fields and summary value fields. Key fields and summary fields use different registration functions in the simple API, however the registered callback functions do not have to be different. In Example 8.16, the same function, `rec_bpp`, is used to compute the bytes-per-packet ratio for a flow record for use in binning records by a key and in proposing candidate values for the summary value.

```
def rec_bpp(rec):
    return int(round(float(rec.bytes) / float(rec.packets)))

register_int_field('bpp', rec_bpp, 0, (1<<32) - 1)
register_int_max_aggregator('maxbpp', rec_bpp, (1<<32) - 1)
```

Example 8.16: `bpp.py`

In Example 8.17, command 2 uses the Python file, `bpp.py`, to create a key field for binning records. Command 3 creates an summary value field instead. The summary value in the example finds the maximum value of all the records in a bin, but there are simple API calls for minimum value and sum as well. For additional summaries, the analyst can use the advanced API function, `register_field`.

```
<1>$ rwfilter --type=in --start-date=2015/06/02 \
   --end-date=2015/06/18 --protocol=0- \
   --max-pass-records=70 --pass=tmp.rw
<2>$ rwuniq tmp.rw --python-file=bpp.py --fields=protocol,bpp \
   --values=records
pro|       bpp|   Records|
 17|        70|         1|
  6|       132|         2|
  6|       410|         1|
 17|        74|         2|
  6|       201|         1|
  6|       116|         1|
 17|        72|         1|
 17|        85|         1|
  6|       409|         1|
  6|       217|         1|
 17|       213|         1|
 17|        84|         1|
  6|       434|         1|
  6|       140|         1|
  6|        40|        15|
  6|       160|         1|
  6|        46|         1|
  6|       236|         1|
  6|        45|        29|
 17|        73|         1|
  6|       174|         5|
  6|       154|         1|
<3>$ rwuniq tmp.rw --python-file=bpp.py --fields=protocol \
   --values=maxbpp
pro|    maxbpp|
  6|       434|
 17|       213|
```

Example 8.17: Calling `bpp.py`

As shown in this chapter, PySiLK simplifies several previously difficult analyses, without requiring coding large scripts. While the programming involved in creating these scripts has not been described in much detail, the scripts shown (or simple modifications of these scripts) may prove useful to analysts.

This page intentionally left blank.

Chapter 9

Tuning SiLK for Improved Performance

While the SiLK tool suite offers a lot of expressive power and can accommodate a wide variety of analyses, processing network flow records can take a long time. This delay comes from retrieving a large number of records and analyzing those records through a complex series of steps. The time it takes to perform an analysis can be reduced by exploiting features of the SiLK suite, the data being processed, and the processing environment.

Upon completion of this chapter you will be able to

- develop a plan to reduce analysis time
- exploit features of the processing environment to reduce analysis time
- exploit data characteristics to reduce analysis time
- employ features of the SiLK suite to reduce analysis time

Hint 9.1: Response Times and Processor Architectures

This chapter presents a series of examples showing the various strategies, with observed response times. Variations on host load, processor and disk configuration, storage organization, repository structure, repository file sizes, and operating system versions all will affect the time values. The performance gain for any strategy cannot be guaranteed for all data sets. However, these techniques do allow retrieval and filtering to be done faster on modern multiprocessor architectures.

The specific results presented here were obtained on a sixteen-processor shared-memory architecture, with two cores per processor. Other processor architectures likely will produce differing amounts of improvement. The times in this chapter are presented to compare and illustrate response time improvements and may not represent the expected times for your environment.

9.1 Introduction

This chapter is organized around an extended example of strategies to improve analysis response time. For each strategy, it compares a non-optimized analytic with an optimized one. While not all of these techniques will be appropriate everywhere, some of them are likely to be applicable to your environment.

9.1.1 Example: Reducing the Run Time of SiLK Analyses

Su Lin, an analyst at a large ISP, has been asked to track down evidence of data transfer using transport protocols that are not TCP, UDP, or ICMP. She needs to run this analysis every day over the course of a month. She would like to run it overnight, while other processing is normally low, and have the results ready by 8:00 am the next morning.

Her first attempt involved sequentially running her analysis on each of 5,000 IP addresses across the last day's data. It took about 30 hours to complete—far too long to run it every evening! She's therefore decided to do a series of short tests to find out which analysis methods might yield better performance.

Su Lin will examine different ways to reduce the time it takes to perform her analysis. In many of her examples, output files are silently deleted (omitted for page formatting in the handbook) between calls to prevent over-writing existing files.

9.1.2 Processor Contention, Data Contention, and Performance

Two processing attributes bound the performance gains that Su Lin seeks, particularly for strategies involving parallelization.

- *Processor contention* is when the number of running processes exceeds the available processor resources, forcing the operating system to delay (wait for an open processor) or time-slice (force temporary waits to share processors) execution. This increases response time and limits the improvement due to parallelization.

- *Data contention* is when multiple processes are competing for transfer of data from the same storage resource (either disk, the network interface, or less commonly, memory). In data contention, one process using the resource blocks its availability to other processes, forcing them to wait. This increases the response time and limits the improvement due to parallelization. While modern disk and network interfaces support some degree of sharing, there is still some waiting involved.

For many SiLK tools, particularly `rwfilter`, `rwsort`, `rwuniq`, and `rwstats`, data contention is a more significant factor on most architectures than processor contention—a phenomenon known as *being I/O bound*.

Su Lin's strategies for improving the performance of her analysis depend on reducing the impact of being I/O bound. She needs to organize the concurrent queries to use different storage resources. For SiLK queries, this means accessing types individually to pull the separate storage in parallel. While the separate access does not completely eliminate being I/O bound, it does provide more gain in response times.

9.2 Using Concurrent `rwfilter` Calls to Spread the Load Across Processors

One approach that Su Lin can take to improve the performance of her `rwfilter` queries is to make more efficient use of the available processing resources. She can structure her queries in parallel to keep the processor busy, in a way that provides results concurrently on separate data. She can then combine the parallelized results to produce her final results.

9.2.1 Parallelizing `rwfilter` Calls By Type

Su Lin can spread the processing load by setting up parallel `rwfilter` calls for each pertinent type of network flow. The operating system on the host machine for the analysis may allocate each parallel call to a separate processor, which spreads the overall load. She can then call the `wait` shell command to assure that all of the parallel calls complete before processing proceeds. (Another way of spreading the load among processors is to use pipes as described in Section 9.7.)

Su Lin also can apply her knowledge of SiLK data types and networking to improve the performance of her analysis. For example, she knows that web traffic uses the TCP protocol and can therefore eliminate the SiLK web types `inweb` and `outweb` from her data pull to save time. This is known as *type dropping*.

Example 9.1 shows the performance improvement that Su Lin achieves by applying these strategies.

1. The naive `rwfilter` call in Command 1 shows the processing time without parallelism by pulling records of all types that are not ICMP, TCP, or UDP flows in a single call. It retrieves them in a little over 8.5 seconds.

2. Commands 2 through 6 pull the same data as separate parallel calls, one for each pertinent type. This version also drops inapplicable types—specifically `inweb` and `outweb`, which are all TCP data and thus irrelevant to this analysis. This type dropping and parallelism cuts the response time by roughly 68%, to a little under 3.2 seconds.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=all --fail=oddproto.raw

real    0m8.539s
user    0m7.106s
sys     0m1.424s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in --fail=oddproto2a.raw &
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out --fail=oddproto2b.raw &
<4>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=int2int --fail=oddproto2c.raw &
<5>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=ext2ext --fail=oddproto2d.raw &
<6>$ wait

real    0m3.188s
user    0m5.476s
sys     0m0.823s
```

Example 9.1: Using Multiple `rwfilter` Processes to Parallelize by Type

9.2.2 Parallelizing `rwfilter` Calls By Flow Time

The SiLK repository is organized by date and hour as well as by type. Instead of parallelizing by type, Su Lin can parallelize by the time of the flow. She will set up a series of `rwfilter` calls to pull separate repository files by start time. Her analysis and the resulting performance improvement are shown in Example 9.2.

1. Command 1 calls `rwfilter` to pull flow data with types `in`, `out`, `int2int`, and `ext2ext` for the time period starting on June 15 and ending on June 19. This command takes 5.569 seconds to execute.

2. Command 2 uses a Bash `for` loop to iterate through the repository files by time. It breaks the time period up into 6-hour bins and calls `rwfilter` on the flow records in each bin as a separate, parallel process.

   (a) The `for` loop first calls the `rwfglob` command to generate a list of repository files. This command takes the same selection parameters as `rwfilter` (in this case, `--start-date`, `--end-date`, and `--type`) and sends a list of files that match those parameters to standard output.

   (b) The loop checks to see if the resulting file contains any data before calling `rwfilter` on it. Some time bins will not have the data that Su Lin seeks.

   (c) The command `mktemp` provides a unique output file name for each `rwfilter` call, all of which are run concurrently.

   (d) The `rwfilter` calls in the loop have the same parameters as the initial call in Command 1, except that they are run on the 6-hour time periods defined by the `for` loop.

3. When the `for` loop in Command 2 ends, the `wait` command pauses until all of the separate `rwfilter` processes complete (similar to Example 9.1).

Overall, Command 2 takes 1.429 seconds to execute—a 73% time savings over the initial `rwfilter` call in Command 1.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/15 --end=2015/06/19 --proto=1,6,17 \
   --type=in,out,int2int,ext2ext --fail=oddproto.raw

real    0m5.569s
user    0m5.294s
sys     0m0.272s
<2>$ for d in 15 15 17 18 19; do \
     for h in 0 6 12 18; do \
       let eh=h+5; \
       flist=$(rwfglob --start-date=2015/06/${d}T$h \
--end-date=2015/06/${d}T$eh --type=in,out,int2int,ext2ext \
--no-summary ); \
       flen=${#flist}; \
       if [ $flen -gt 0 ]; then \
         OUTNAME=$(mktemp -u --suffix=".raw" oddprotoXXX); \
         rwfilter $flist --proto=1,6,17 --fail=$OUTNAME & \
         fi; \
       done; \
     done; \
wait

real    0m1.429s
user    0m3.638s
sys     0m0.291s
```

Example 9.2: Using Concurrent `rwfilter` Processes by Hour

Help with `rwfglob`. For a list of command options, type `rwfglob --help` at the command line. For more information about the `rwfglob` command, type `man rwfglob`.

Visualizing parallelized `rwfilter` calls. Figure 9.1 shows the time improvement that Su Lin gains through parallelizing her initial `rwfilter` call by time, graphed using Microsoft Excel. (Her analysis found no data on June 15 and 19, so these days are not included in the graph.)

- The time to run the initial `rwfilter` call with no parallelization (Command 1 in Example 9.2) is plotted at the top of the graph. This `rwfilter` call processed 1619 flow record files from the SiLK repository.
- The parallelized calls to `rwfilter` (Command 2 in Example 9.2) are plotted beneath the initial, non-parallelized `rwfilter` call. Each call to `rwfilter` launches as the `for` loop runs. The runtime for each parallelized call depends on how much data was captured during that 6-hour time bin. The file count for each call shows how many flow files were retrieved during that time period.

Out of curiosity, Su Lin adds up the total time it would have taken to run the parallelized `rwfilter` commands sequentially: 6.083 seconds. It's longer than the time it took to execute the `rwfilter` call in Command 1 because it includes the time to execute the loop. Even with the overhead of the `for` loop, executing these `rwfilter` commands in parallel saves a significant amount of time.

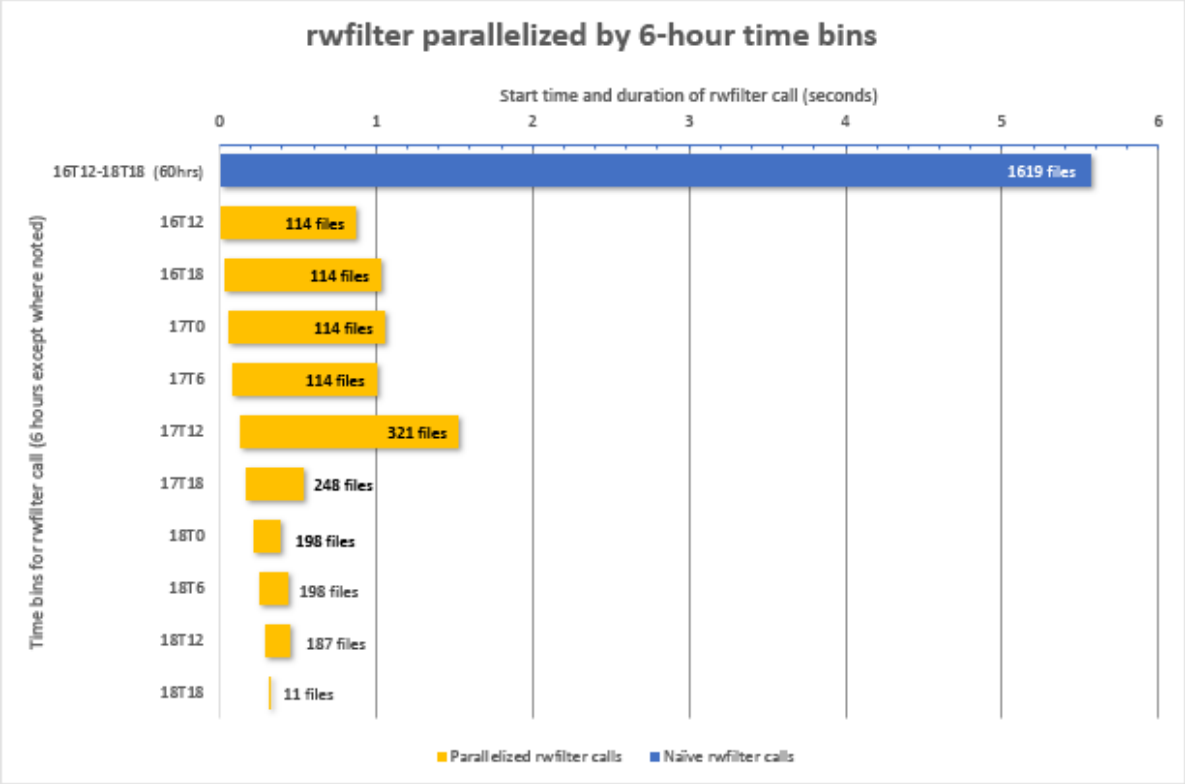

Figure 9.1: Response times for `rwfilter` parallelized by 6-hour time bins

9.3 Combining Results From Concurrent `rwfilter` Calls via `rwuniq`, `rwcount`, and `rwstats`

After all concurrent `rwfilter` commands are done executing, Su Lin must draw the results together for analysis. Unfortunately, due to the way that SiLK stores files, the presence of flows in specific files may not align in the way she expects. This causes irregularities in cumulative volumes and trend lines. Processing the files separately rather than as a single result magnifies these irregularities.

To minimize this problem, Su Lin can bring the files together to generate results with `rwcount`, `rwuniq`, or `rwstats` before plotting the data. Example 9.3 shows her strategies for combining her results.

1. Command 1 calls `rwfilter` to retrieve records from the repository, `rwsort` to order them, then `rwuniq` to generate a profile (which in this case is sent to `/dev/null`, discarding it automatically). The command runs in a little over 6.1 seconds.
2. Command 2 does the same thing as Command 1, but uses the `--presorted` parameter for `rwuniq`, indicating `rwuniq` can be more memory-efficient in its processing. This command runs in about the same time as Command 1.
3. Command 3 skips the call to `rwsort`, letting `rwuniq` handle the ordering of the data. This command runs just slightly slower than Command 1.
4. Command 4 skips the `rwfilter` call, letting `rwuniq` open previously retrieved flow record files and produce a merged result. This command runs much faster, taking a bit over 0.8 seconds to complete.

While a merged input file executes more slowly than letting the profiling tools merge the files, it may be desirable for archival purposes. In this case, consider using `rwsort` because it both merges and orders the flow records. Merging data with `rwsort` is described further in Section 9.5.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in,out,int2int,ext2ext --fail=stdout \
| rwsort --fields=dip \
| rwuniq --fields=dip --values=bytes,packets \
   --output=/dev/null

real    0m6.166s
user    0m5.503s
sys     0m0.683s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in,out,int2int,ext2ext --fail=stdout \
| rwsort --fields=dip \
| rwuniq --fields=dip --values=bytes,packets --presorted \
   --output=/dev/null

real    0m6.181s
user    0m5.445s
sys     0m0.738s
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in,out,int2int,ext2ext --fail=stdout \
| rwuniq --fields=dip --values=bytes,packets \
   --output=/dev/null

real    0m6.202s
user    0m5.497s
sys     0m1.595s
<4>$ rwuniq --fields=dip --values=bytes,packets oddproto2a.raw \
   oddproto2b.raw oddproto2c.raw oddproto2d.raw \
   --output=/dev/null

real    0m0.837s
user    0m0.050s
sys     0m0.770s
```

Example 9.3: Response Time for Sorting Records and File Parameters

9.4 Parallelizing via the `rwfilter --threads` Parameter

Another parallelization strategy that Su Lin can pursue is to request that each `rwfilter` call operate using lightweight parallel threads. The `--threads` parameter causes `rwfilter` to spawn the number of parallel threads specified by its argument. Each thread is allocated an input file to filter. While the output is not parallelized (as it is with concurrent `rwfilter` commands) the input, parsing, and filtering of records is parallelized.

9.4.1 Improving `rwfilter` Performance with `--threads`

Example 9.4 shows an example of the performance boost that Su Lin can realize by using the `rwfilter --threads` parameter.

1. Command 1 is the same naive `rwfilter` command as in the previous examples, again taking a little over 8.5 seconds to complete.
2. Command 2 shows the same query, this time with `--threads=4`, which decreases the response time to around 3.4 seconds.
3. In Command 3, increasing the argument to `--threads=8` decreases the response time still further, to around 2.5 seconds.
4. In Commands 4 and 5, further increases to `--threads=12` and `--threads=16` decreases the response time by smaller amounts, to around 2.4 and 2.2 seconds, respectively.

Figure 9.2 shows the response time for each `rwfilter --threads` value: one thread (the default, naive `rwfilter` call), four threads, eight threads, 12 threads, and 16 threads. (These results were graphed using Microsoft Excel.)

9.4.2 Effect of `--threads` On Other `rwfilter` Parameters

The use of `--threads` affects some of the other `rwfilter` parameters. With `--threads`, the `--max-pass` and `--max-fail` limits may be exceeded. This excess on the maximum output also affects the `--print-statistics` and `--print-volume-statistics` parameters if they are used. For many analysts, however, these drawbacks are worth the improvement in performance offered by `--threads`.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --type=all \
   --proto=1,6,17 --fail=oddproto.raw

real    0m8.568s
user    0m7.157s
sys     0m1.402s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --type=all \
   --proto=1,6,17 --threads=4 --fail=oddproto2.raw

real    0m4.011s
user    0m9.646s
sys     0m1.583s
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --type=all \
   --proto=1,6,17 --threads=8 --fail=oddproto3.raw

real    0m2.845s
user    0m9.818s
sys     0m1.651s
<4>$ rwfilter --start=2015/06/01 --end=2015/06/30 --type=all \
   --proto=1,6,17 --threads=12 --fail=oddproto4.raw

real    0m2.258s
user    0m7.952s
sys     0m1.760s
<5>$ rwfilter --start=2015/06/01 --end=2015/06/30 --type=all \
   --proto=1,6,17 --threads=16 --fail=oddproto5.raw

real    0m2.166s
user    0m8.423s
sys     0m1.809s
```

Example 9.4: Using `rwfilter --threads` to Reduce Response Time

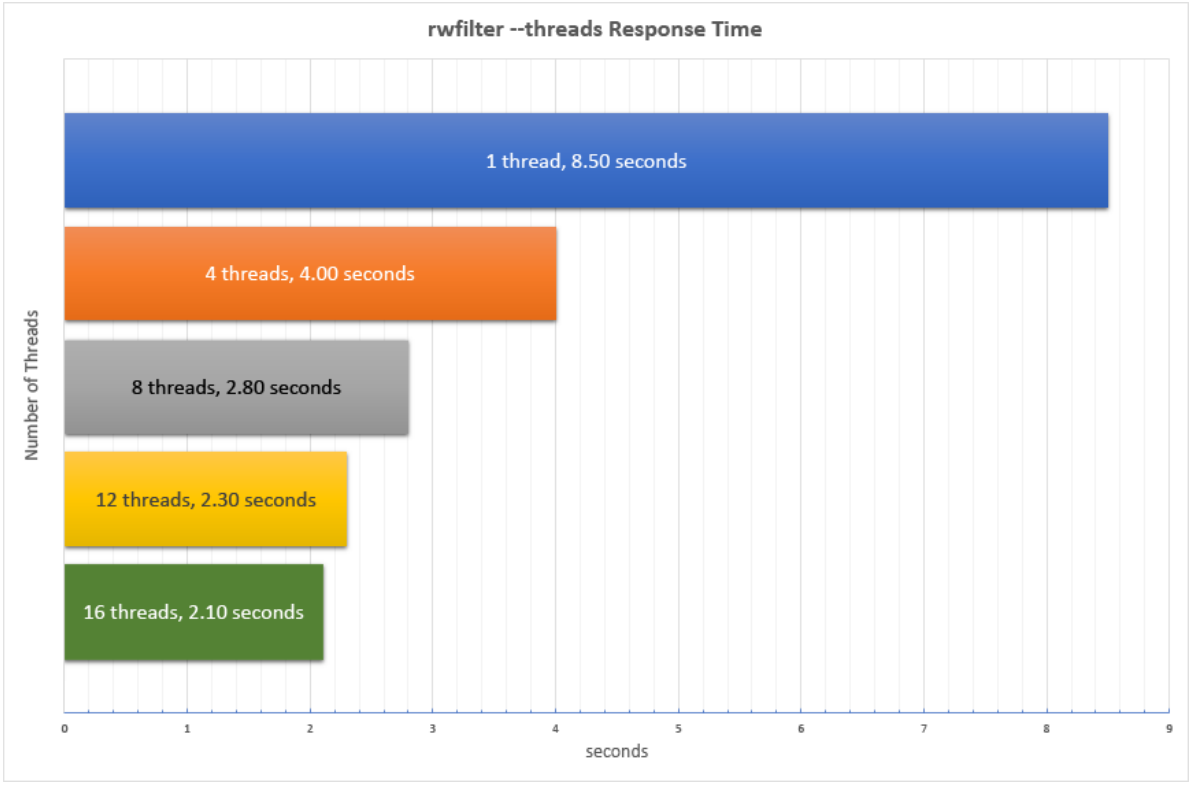

Figure 9.2: Comparing response times for `rwfilter --threads` values

9.4.3 Limitations On `--threads` Performance Improvements

Unfortunately, Su Lin soon discovers that the performance improvement due to use of threads decreases with the number of concurrent processes (such as when multiple analysts are running `rwfilter` with `--threads`). Figure 9.3 shows this decrease.

- For a single process, the performance improvement is as described earlier in this section.
- For four concurrent `rwfilter` processes (in this case, four identical processes, representing four users making equivalent `rwfilter` calls), `--threads=1` (the default) takes about 9.6 seconds. Going to `--threads=4` with four processes improves the response time to approximately 4.5 seconds (a gain of 53% vs 60% for the single process). Going to `--threads=8` improves it to approximately 3.9 seconds (a total gain of 59% versus 71% for the single process). Going to 12 and 16 threads in four processes improves it to approximately 3.7 and 3.6 seconds, respectively (total gains of 61% and 62% versus 72% and 74% for the single process).
- For ten concurrent processes, `--threads=4` improves the response time by 35%, while going to `--threads=16` only improves it by 46%.

These figures indicate that when multiple users share the same storage and processing resources, the amount of improvement due to threads above four may not be of much benefit.

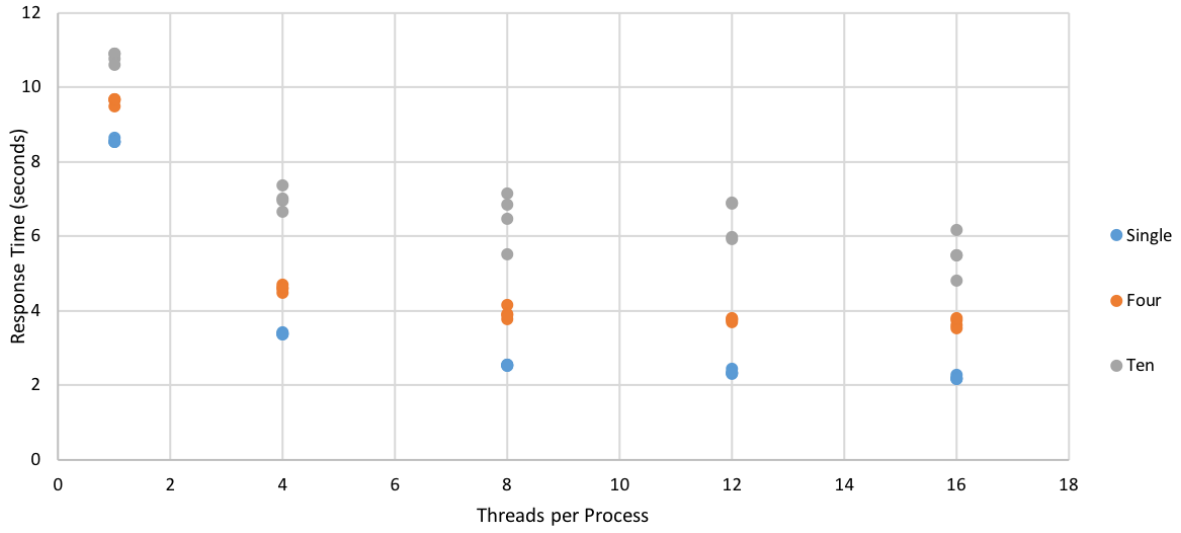

Figure 9.3: Decreasing Response Time by Using `rwfilter --threads` Profiled by Number of Running Processes

The curve shown in Figure 9.3 reflects a common experience in using parallelization to improve performance. This experience is codified as Amdahl's law (after Gene Amdahl, who presented it at the 1967 AFIPS Fall Joint Computer Conference[30]). It states that the performance gain by parallelization is limited by the

[30] Amdahl, Gene M. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities." 1967. AFIPS Conference Proceedings (30): 483–485. doi:10.1145/1465482.1465560. [Accessed May 14, 2020] http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

sequential portions of the program. Amdahl's law predicts that performance gain will be most dramatic among the first few parallel threads. After that, the gain will decrease rapidly until no improvement results.

9.5 Constructing Efficient Queries

Su Lin can improve the performance of SiLK by designing her analytic to execute more efficiently. This often takes the form of describing key characteristics of the traffic being sought, including:

- the type of traffic wanted (`in`, `out`, `inweb`, `outweb`, `int2int`, `ext2ext`)
- the transport protocols wanted (or not wanted)
- packet sizing (at least bounding the byte count or packet count by maximum or minimum)
- timing for the traffic (either in terms of start time or of duration)
- if TCP, whether this is the initiating side or responding side of the traffic (indicated by initiating TCP flags) and whether they are complete or partial TCP sessions (indicated by terminating TCP flags)
- IP address ranges of interest (either on the internal or external ends)
- whether the traffic represents a single interaction or multiple interactions
- whether the result desired is a count, a frequency, or some statistical summary
- whether IPv6 data is useful for this analysis

Hint 9.2: Performance for IPv4 Versus IPv6 Addresses

IPv4 data is generally processed more quickly than IPv6 data. If IPv6 addresses are not useful for your analysis but SiLK has been configured for IPv6, you can force SiLK to generate IPv4 data by doing one of the following:

- Use the following Bash command to override SiLK's IPv6 configuration (i.e., the `SILK_IPV6_POLICY` environment variable):

  ```
  export SILK_IPV6_POLICY=ignore
  ```

- Specify the `--ipv6-policy=ignore` parameter to make individual SiLK commands generate IPv4 data instead of IPv6 data.

This section describes strategies that Su Lin can use to construct more efficient queries.

9.5.1 Pipelining Calls to SiLK commands

The characteristics listed above in Section 9.5 often form the basis of added partitioning parameters on calls to `rwfilter` or added thresholds for `rwuniq` or `rwstats`. Su Lin can pipeline multiple invocations of these

tools together for efficiency. In practice, `rwfilter` usually has the longest response time. To work around this issue, she can structure the analysis as a series of calls to `rwuniq` or `rwstats` that explore data that's already been retrieved by `rwfilter` instead of pulling fresh data from the repository at every step.

Example 9.5 shows the response time improvement from minimizing calls to `rwfilter` and using pipelining between SiLK commands instead of writing data to disk. Su Lin applies the concepts behind the `rwfilter` manifold described in Section 4.2.1, which can also improve performance by pipelining successive calls to `rwfilter` instead of writing files to disk at each step.

1. Command 1 shows a series of calls that use `rwfilter` to pull the same data three times. Each `rwfilter` call feeds into a `rwuniq` call to profile the data by source address, by destination address, and by start time. This series of calls takes just over 17 seconds to run.

2. Command 2 shows a series of calls that does the same thing as Command 1, but only calls `rwfilter` once. it stores the result in a file that is then profiled by the three `rwuniq` calls. This series of calls takes just over 6 seconds to run, showing the impact of avoiding repeated `rwfilter` calls.

3. Command 3 shows a pipelined series of calls that does the same thing as Command 1, but passes the data via pipes from a single `rwfilter` call to the three `rwuniq` calls. It links the `rwuniq` calls with `--copy-input=stdout` to pass all of the flows to each `rwuniq` instance. This series of calls takes just over 5.3 seconds to run, reflecting the savings due to pipelining rather than writing and reading a file.

9.5.2 Using Flow Characteristics To Improve `rwfilter` Efficiency

Su Lin can take advantage of different flow characteristics to more closely define the scope of `rwfilter` queries, improving their speed and efficiency. Her goal is to more efficiently identify possible file transfer using uncommon protocols from the internal network to external IP addresses. To do this, she progressively refines her `rwfilter` query to concentrate only on records that show communication between internal and external hosts via uncommon protocols. Her analysis is shown in Example 9.6.

1. Command 1 displays the naive approach, paralleling the one used in preceding examples: find all uncommon protocol usage with `rwfilter`, then use `rwuniq` to aggregate bytes transferred for each destination IP address. The advantage of this approach is that it is simple to write. Its disadvantage is that it could retrieve records that are not data transfers and that destination addresses are not necessarily external addresses. This command takes a little over 8.6 seconds.

2. For this data, Command 2 shows that the output contains the header line and entries for nine addresses, three of which were found to be internal addresses. These addresses are irrelevant to Su Lin's inquiry and can be eliminated from the `rwfilter` query to save time.

3. Command 3 displays a revised approach: find all outbound uncommon protocol usage, then aggregate bytes transferred in flows of three packets or more to each external IP address (the destination address for an outgoing flow). This approach has the advantage of still being fairly simple to write with fewer uninteresting records included in results. Its disadvantage is that it may will miss low-and-slow transfers that divide into separate flows, and might include addresses that receive occasional bursts of packets containing protocol signals rather than file transfer. This command took a bit over 3.5 seconds to execute.

4. Command 4 shows that this query identified six external addresses and no internal addresses.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,int2int,ext2ext --fail=stdout \
| rwuniq --fields=dip,protocol \
   --values=bytes,flows,distinct:sip \
   --output=oddproto-dip.txt ; \
rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,int2int,ext2ext --fail=stdout \
| rwuniq --fields=sip,protocol \
   --values=bytes,flows,distinct:dip \
   --output=oddproto-sip.txt ; \
rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,int2int,ext2ext --fail=stdout \
| rwuniq --fields=stime,protocol --bin-time=3600 \
   --values=bytes,flows --output=oddproto-stime.txt

real    0m17.005s
user    0m13.658s
sys     0m5.043s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,int2int,ext2ext --fail=oddproto.raw ; \
rwuniq --fields=dip,protocol \
   --values=bytes,flows,distinct:sip oddproto.raw \
   --output=oddproto-dip.txt ; \
rwuniq --fields=sip,protocol \
   --values=bytes,flows,distinct:dip oddproto.raw \
   --output=oddproto-sip.txt ; \
rwuniq --fields=stime,protocol --bin-time=3600 \
   --values=bytes,flows oddproto.raw \
   --output=oddproto-stime.txt

real    0m6.013s
user    0m4.787s
sys     0m1.209s
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,int2int,ext2ext --fail=stdout \
| rwuniq --fields=dip,protocol \
   --values=bytes,flows,distinct:sip \
   --output=oddproto-dip.txt --copy-input=stdout \
| rwuniq --fields=sip,protocol \
   --values=bytes,flows,distinct:dip \
   --output=oddproto-sip.txt --copy-input=stdout \
| rwuniq --fields=stime,protocol --bin-time=3600 \
   --values=bytes,flows --output=oddproto-stime.txt

real    0m5.351s
user    0m4.758s
sys     0m1.318s
```

Example 9.5: Avoiding multiple `rwfilter` Commands to Increase Performance

5. Command 5 shows another improved approach: find outgoing uncommon protocol traffic with byte counts that indicate more than header information, then aggregate bytes transferred to each external IP address, presenting only aggregates greater than the threshold count of bytes transferred. The advantage to this approach is that it uses a more robust recognition of file transfer that would allow identification of low-and-slow transfers. Its disadvantage is the need to calibrate thresholds properly. This command also took a bit over 3.5 seconds to execute.

6. Command 6 shows that it found five external addresses. Examination of these addresses found that they were all recipients of tunneled traffic, either VPN or OSPF tunnels.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=all --fail=stdout \
| rwuniq --fields=dip --values=bytes \
   --output=dip-bytes.txt

real    0m8.624s
user    0m7.122s
sys     0m1.509s
<2>$ wc -l dip-bytes.txt
10 dip-bytes.txt
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,ext2ext --fail=stdout \
| rwuniq --fields=dip --values=bytes,packets --packets=3- \
   --output=dip3-bytes.txt

real    0m3.550s
user    0m3.197s
sys     0m1.071s
<4>$ wc -l dip3-bytes.txt
7 dip3-bytes.txt
<5>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,ext2ext --fail=stdout \
| rwfilter stdin --bytes-per=65- --pass=stdout \
| rwuniq --fields=dip --values=bytes,packets --bytes=2000- \
   --output=dipb-bytes.txt

real    0m3.525s
user    0m3.203s
sys     0m1.719s
<6>$ wc -l dipb-bytes.txt
6 dipb-bytes.txt
```

Example 9.6: Closely Defining Analysis Problem to Increase Performance

9.5.3 Specifying Fewer SiLK Types In `rwfilter` Calls

One reason for the performance improvement in Example 9.6 was that Su Lin's `rwfilter` queries targeted the SiLK types relevant to the behavior she is investigating. Instead of using `--types=all`, which references

six types (`in`, `out`, `inweb`, `outweb`, `int2int`, and `ext2ext`), Su Lin limited her queries to just two types for outbound non-TCP traffic (`out` and `ext2ext`). This caused `rwfilter` to open fewer parts of the repository and thus execute more quickly.

Example 9.7 isolates the effect of reducing the number of SiLK types in a `rwfilter` call.

1. Command 1 shows the naive query used in previous examples, which takes 8.591 seconds to execute.
2. The `rwfilter` call in Command 2 uses a more restrictive `--type` parameter that only pulls records with four SiLK flow types (`--type=in,out,int2int,ext2ext`). The execution time falls to 6.13 seconds, an improvement of approximately 29%.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=all --fail=oddproto.raw

real    0m8.591s
user    0m7.067s
sys     0m1.516s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in,out,int2int,ext2ext --fail=oddproto2.raw

real    0m6.130s
user    0m5.435s
sys     0m0.687s
```

Example 9.7: Using Only Needed `rwfilter` Types to Increase Performance

Figure 9.4 shows the time improvement that Su Lin gains through type dropping (graphed using Microsoft Excel).

9.5.4 Constraining `rwfilter` Output

An additional way to improve the response time for `rwfilter` lies in exploiting the fact that it is more I/O bound than compute bound. Su Lin can use partitioning parameters such as `--type`, `--packets`, or `--bytes` to constrain the output that `rwfilter` generates. She can also use the `--start` and `--end` parameters to limit `rwfilter` output by time period. This is somewhat counter-intuitive: longer and more complex `rwfilter` commands often run faster than shorter and simpler ones.

In Example 9.8, Su Lin contrasts a naive `rwfilter` command (Command 1) against a more complex `rwfilter` command (Command 2) that uses both a more restrictive `--types` parameter and additional `--packets` and `--bytes` parameters. The response time improves from 8.5 seconds to just over 6 seconds.

9.5.5 Merging the Results of Multiple `rwfilter` Calls

It can be difficult to produce an efficient single `rwfilter` command that pulls all of the data of interest. In some analyses, several calls to `rwfilter` are needed. Examples of when multiple commands are useful include:

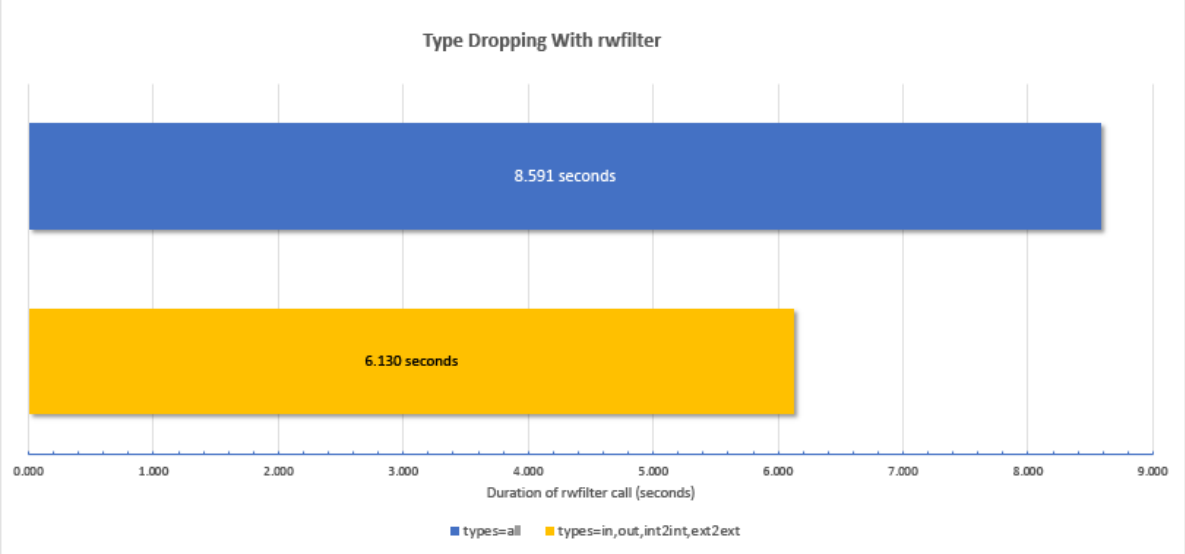

Figure 9.4: Response times for `rwfilter` with all SiLK types and limited SiLK types

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=all --fail=oddproto.raw

real    0m8.623s
user    0m7.103s
sys     0m1.511s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 \
   --proto=50,51,83 --type=in,out,int2int,ext2ext \
   --packets=3- --bytes=400- --pass=oddproto2.raw

real    0m6.096s
user    0m5.419s
sys     0m0.671s
```

Example 9.8: Using Additional Parameters with `rwfilter` to Increase Performance

- comparing activity during an incident with activity a week earlier and a week later
- comparing activity on an affected network segment against activity on an unaffected one
- identifying progress over time of activity indicative of a particular group of intruders
- identifying the impact of fielding a new defensive measure or modifying a service to improve security

In these cases, Su Lin can pull data separately for the subsets of traffic she desires for comparison or identification. She can then merge her results to form a common pool of data for computing statistical summaries and trends.

SiLK provides several ways to do this merging. Three tools that are particularly useful are `rwcat`, `rwappend`, and `rwsort`.

- `rwcat` concatenates the series of flow records in each of its input files into a single series for output, which is most useful for analyses where the order of records in not important.
- `rwappend` inserts the series of flow records in each file (in order) into the end of the output file, which is most useful for analyses where the data is already in the desired order.
- `rwsort` does a merge-sort of the records in all of its input files to produce a single ordered output file, which is most useful when the analyst either cannot depend on the order of the data, or needs to impose a specific order.

These tools have already been introduced and are discussed more thoroughly in Sections 2.2.7 (for `rwsort`) and 6.2.4(for `rwappend` and `rwcat`).

Example 9.9 shows how Su Lin uses these three commands to merge separate flow files.

1. Commands 1-5 generate four separate flow files.
2. Command 6 shows how to use `rwcat` to simply concatenate the flow files. This takes approximately 0.01 seconds for these data, but the data will be in no particular order.
3. Command 7 shows how to use `rwappend` to perform a similar amalgamation of the records in the four input files, taking approximately the same time as the `rwcat` command. However, the output file records are in the order of the input files.
4. Command 8 shows how to use `rwsort` to merge the four input files. This takes about the same amount of time as the `rwcat` and `rwappend` commands. However, the records from all files are put in time order by start time on the output.

The distinction between these three commands is based on what can be assumed about the semantics of ordering, rather than which command runs fastest. In particular, if Su Lin needs records that are stored in a dependable order, she will use `rwsort` to merge them. It takes approximately the same amount of time as the other methods and produces a file that she will not have to sort later!

```
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=in --fail=oddproto2a.raw &
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out --fail=oddproto2b.raw &
<3>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=int2int --fail=oddproto2c.raw &
<4>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=ext2ext --fail=oddproto2d.raw &
<5>$ wait
<6>$ rwcat oddproto2a.raw oddproto2b.raw oddproto2c.raw \
   oddproto2d.raw --output=oddproto.raw

real    0m0.012s
user    0m0.003s
sys     0m0.002s
<7>$ rwappend --create=oddproto2a.raw oddproto.raw \
   oddproto2a.raw oddproto2b.raw oddproto2c.raw \
   oddproto2d.raw

real    0m0.013s
user    0m0.001s
sys     0m0.005s
<8>$ rwsort --fields=stime oddproto2a.raw oddproto2b.raw \
   oddproto2c.raw oddproto2d.raw --output=oddproto.raw

real    0m0.016s
user    0m0.005s
sys     0m0.004s
```

Example 9.9: Combining Flow Files with `rwcat`, `rwappend`, and `rwsort`

9.6 Using Coarse Parallelism

Flow record files can be quite large and Su Lin knows from experience that processing a very large file can be slow. The SiLK tools need to use paging (moving data in and out of memory to disk) to accommodate very large files. It may be faster to divide these large flow record sizes into manageable pieces using a command such as `rwsplit`, then parallelize the pieces with separate commands.This process is known as *coarse parallelism* since it exploits larger features of the data to work concurrently.

Su Lin wants to find out if splitting up her files with `rwsplit` to perform coarse parallelism will improve the performance of her analysis. Her analytic is shown in Example 9.10 .

1. The `rwfilter` call in Command 1 pulls a large amount of data (all of the TCP, UDP, and ICMP flows in the data set), then uses `rwsort` to aggregate source and destination addresses. It then invokes `rwuniq` to profile possible file transfers. This information can be used to give context to results for less commonly-used protocols. This command takes a bit over 32.7 seconds.

2. Command 2 starts with similar calls to `rwfilter` and `rwsort`. It then calls `rwsplit` to divide the records into several files. Since the `rwuniq` command summarizes by source and destination IP address, this `rwsplit` command divides the records into portions governed by the number of IP addresses. This produces three parts, each of which is then summarized in parallel with the others.

   This command takes a little over 37.5 seconds to execute. This is a bit longer than Command 1, since this data set is too small to require paging of memory even when pulling all of the relevant data. However, this method may result in shorter response times for larger datasets than the one Su Lin uses for her tests.

3. Command 3 shows the length of the result files, which, accounting for headings one each file, indicates one address is split across two files.

4. By using a UNIX text sort, the files can be brought together and the duplication dealt with, as shown in Command 4.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,ext2ext --pass=stdout \
| rwsort --fields=dip,sip \
| rwuniq --fields=dip --values=bytes,packets --bytes=400- \
   --packets=3- --output=comproto.txt

real    0m24.942s
user    0m20.899s
sys     0m9.828s
<2>$ rwfilter --start=2015/06/01 --end=2015/06/30 --proto=1,6,17 \
   --type=out,outweb,ext2ext --pass=stdout \
| rwsort --fields=dip,sip \
| rwsplit --ip-limit=800 --basename='comproto' ; \
rwuniq comproto.00000000.rwf --fields=dip --presort \
   --values=bytes,packets --bytes=400- --packets=3- \
   --output=comproto.00000000.txt& \
rwuniq comproto.00000001.rwf --fields=dip --presort \
   --values=bytes,packets --bytes=400- --packets=3- \
   --output=comproto.00000001.txt& \
rwuniq comproto.00000002.rwf --fields=dip --presort \
   --values=bytes,packets --bytes=400- --packets=3- \
   --output=comproto.00000002.txt& \
wait

real    0m34.255s
user    0m26.841s
sys     0m12.725s
<3>$ wc -l comproto*.txt
   689 comproto.00000000.txt
    58 comproto.00000001.txt
   242 comproto.00000002.txt
   986 comproto.txt
  1975 total
<4>$ sort comproto.00000000.txt comproto.00000001.txt \
   comproto.00000002.txt comproto.txt | uniq | wc -l
989
```

Example 9.10: Coarse Parallelism of `rwuniq` using `rwsplit`

9.7 Using Named Pipes and Process Substitution

When flow files get large, writing and reading them from disk can be quite slow. While temporary files are a useful tool in analysis, they can also greatly slow down the analysis process. To avoid this delay, Su Lin can make use of process-to-process communication. UNIX systems have a variety of options for this:

- *Pipelining processes* (pipes for short), where a generating process sends input to a receiving process progressively, either via standard output and input or via specially-created named pipes (as is used in Section 4.1.2). Pipes are used extensively throughout this handbook.
- *Process substitution*, where a command string (possibly involving pipes) is used in place of a file or stream.
- *Sockets*, which are used for either process-to-process or host-to-host communication. While they are very powerful tools for harnessing multiple computers in a single analysis, they are complex to set up and won't be further discussed in this handbook.

Example 9.11 shows how Su Lin uses pipes and process substitution to make her analytic faster and more efficient.

1. Commands 1 through 3 show a naive approach, where all of the possibly-relevant traffic is gathered into a flow record file. The file is refiltered twice to isolate specific cases. Each `rwfilter` call feeds into a `rwuniq` call to generate two profiles: one for transport protocol 50, and the other for transport protocols that are not 50, 1 (ICMP), 6 (TCP) or 17 (UDP). This command takes a little over 17.3 seconds on this data set.
2. Command 4 uses `mkfifo` to create a named pipe.
3. Command 5 pulls records from the repository and filters out records with the ICMP, TCP, and UDP protocols. It then passes the records for transport protocol 50 to the named pipe and the records for other protocols to `rwuniq` to generate a profile.
4. Concurrently, Command 6 pulls the protocol 50 records from the named pipe and generates a similar profile.
5. Command 7 uses `wait` to let Commands 5 and 6 complete. Altogether, this sequence with the named pipes takes a little over 3.5 seconds to complete.
6. Command 8 shows the use of process substitution in place of the named pipe. The first call to `rwfilter` is the same as in Command 5. The second call to `rwfilter` uses process substitution, indicated by the use of `>(`, as the argument to its `--pass` parameter.
7. Following the `>(`, Command 8 calls `rwuniq` with input set to the protocol 50 records produced by the `--pass` and output sent to a file—the same profile produced in previous commands, ending with a close parenthesis. The second `rwfilter` call also uses `--fail=stdout` to send non-protocol-50 records to a final `rwuniq` call to generate a similar profile for those records.

   Altogether, this command sequence takes a little over 3.5 seconds to execute. It does not require the `mkfifo` or `wait` commands; concurrency is handled by the process substitution.
8. Command 9 then counts the lines in all of the output text files, reflecting that each `port50` and `not50` profiles from `rwuniq` calls produced the same output.

```
Show real time, user time, and processor time for each command
<1>$ rwfilter --start=2015/06/01 --end=2015/06/30 \
   --type=out,ext2ext --proto=0- --pass=oddproto.raw
<2>$ rwfilter oddproto.raw --proto=50 --pass=stdout \
| rwuniq --fields=dip --values=bytes,packets --packets=3- \
   --output=dip-port50.txt
<3>$ rwfilter oddproto.raw --proto=1,6,17,50 --fail=stdout \
| rwuniq --fields=dip --values=bytes,packets --packets=3- \
   --output=dip-not50.txt

real    0m17.367s
user    0m13.010s
sys     0m2.270s
<4>$ mkfifo /tmp/proto50.fifo
<5>$ rwfilter --start=2015/06/01 --end=2015/06/30 \
   --type=out,ext2ext --proto=1,6,17 --fail=stdout \
| rwfilter stdin --proto=50 --pass=/tmp/proto50.fifo \
   --fail=stdout \
| rwuniq --fields=dip --values=bytes,packets --packets=3- \
   --output=dip3-not50.txt&
<6>$ rwuniq --fields=dip --values=bytes,packets --packets=3- \
   /tmp/proto50.fifo --output=dip3-port50.txt&
<7>$ wait

real    0m3.559s
user    0m3.257s
sys     0m1.025s
<8>$ rwfilter --start=2015/06/01 --end=2015/06/30 \
   --type=out,ext2ext --proto=1,6,17 --fail=stdout \
| rwfilter stdin --proto=50 --pass=>(rwuniq \
   --fields=dip --values=bytes,packets --packets=3- \
   --output=dip4-port50.txt) --fail=stdout \
| rwuniq --fields=dip --values=bytes,packets --packets=3- \
   --output=dip4-not50.txt

real    0m3.583s
user    0m3.197s
sys     0m0.364s
<9>$ wc -l dip*.txt
   5 dip3-not50.txt
   3 dip3-port50.txt
   5 dip4-not50.txt
   3 dip4-port50.txt
   5 dip-not50.txt
   3 dip-port50.txt
  24 total
```

Example 9.11: Pipes and Process Substitution to Improve Response Time

9.8 Specifying Local Temporary Files

The location of temporary file storage can affect the performance of Su Lin's analysis. Several SiLK commands (in particular, `rwsort`, `rwstats`, and `rwuniq`) have a `--temp-dir` parameter that specifies where temporary files are generated. In general, the default location (`/tmp`, the system temporary partition) works quite well.

However, where this partition is too small or otherwise not suitable for a particular analysis, Su Lin can improve the speed of her analysis by using a local directory on the host where the analysis is being performed instead of a remote or network-mounted directory. This avoids the overhead of reading and writing to disk over a network.

9.9 Administrative Actions to Improve SiLK Performance

While this chapter has focused on strategies that Su Lin can apply as an analyst to decrease the time for running her analyses, the administrators that configure the SiLK installation and repository can also improve the response time of analyses. Strategies that administrators can use include:

- Do not compile the SiLK installation to support IPv6 unless there is IPv6 traffic on their network. Many modern networks, particularly those that handle mobile traffic or embedded devices, will have IPv6 traffic present, so this option should be used with care.
- Remove or comment out sensors that have been dropped, obviated, or otherwise produce no data. Removing them from the `silk.conf` file may improve performance.
- Remove unused SiLK data types (especially from the `default-types` definition) in the `silk.conf` file.

9.10 Summary of Strategies to Improve SiLK Performance

Su Lin now has several strategies for improving the performance of her analysis, including parallelizing `rwfilter` calls, dropping SiLK data types, using pipes and process substitution, and storing temporary files on a local disk. Some of these strategies offer significant time savings over her initial, naive SiLK calls! By combining these strategies, she should be able to reduce the run time for her analysis enough to execute it every evening.

The strategies in this chapter are intended to be a jumping-off point for further exploration and tuning of your analyses. Applying them can speed up the processing of large datasets with SiLK. Over time, you will discover which approaches are most beneficial to your analyses and work best for your hardware and network configuration.

Appendix A

TCP/IP, UDP and ICMP Headers

Transmission Control Protocol/Internet Protocol (TCP/IP) is the foundation of internetworking. All packets analyzed by the SiLK system use protocols supported by the TCP/IP suite.

This appendix describes the format of the IP and TCP headers and the structure of UDP and ICMP messages, all of which are relevant to SiLK analyses. For each protocol, it shows the fields that are recorded by SiLK's data collection tools and lists common services.

A.1 Structure of the IP Header

IP passes collections of data as datagrams. Two versions of IP are currently used: versions 4 and 6, referred to as IPv4 and IPv6, respectively. IPv4 still constitutes the vast majority of IP traffic in the Internet. IPv6 usage is growing, and both versions are fully supported by the SiLK tools. Figure A.1 shows the breakdown of IPv4 datagrams. Fields that are not recorded by the SiLK data collection tools are grayed out. With IPv6, SiLK records the same information, although the addresses are 128 bits, not 32 bits.

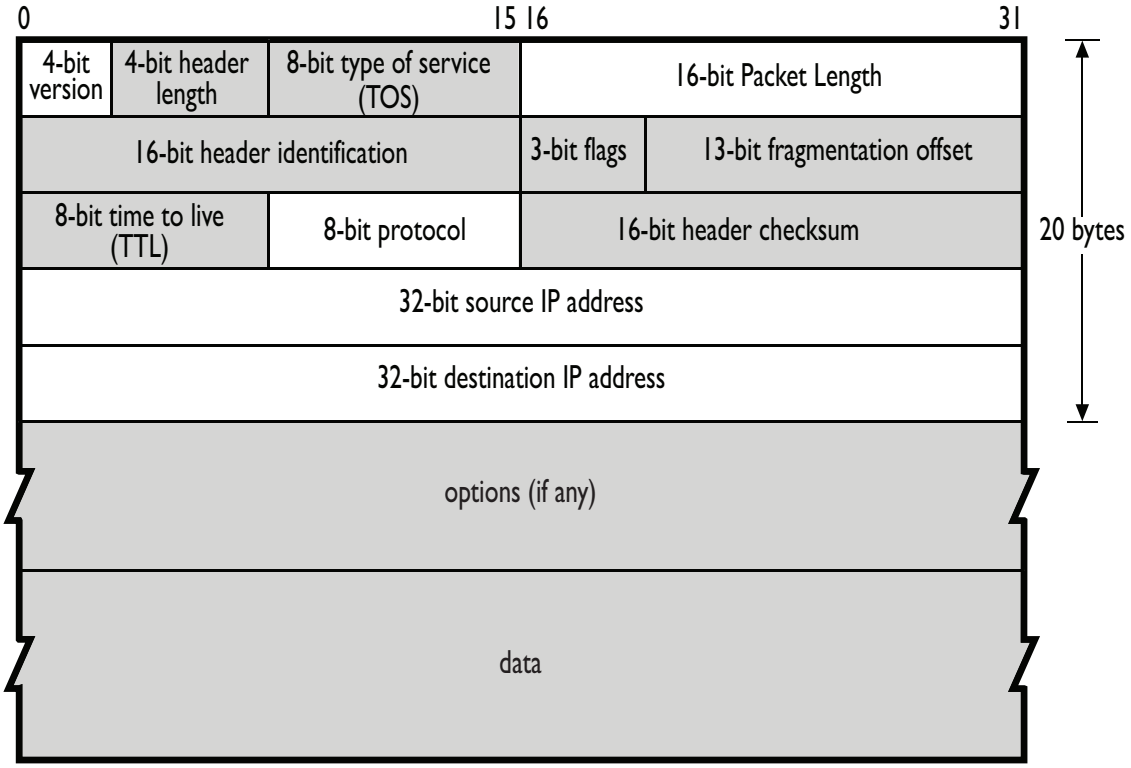

Figure A.1: Structure of the IPv4 Header

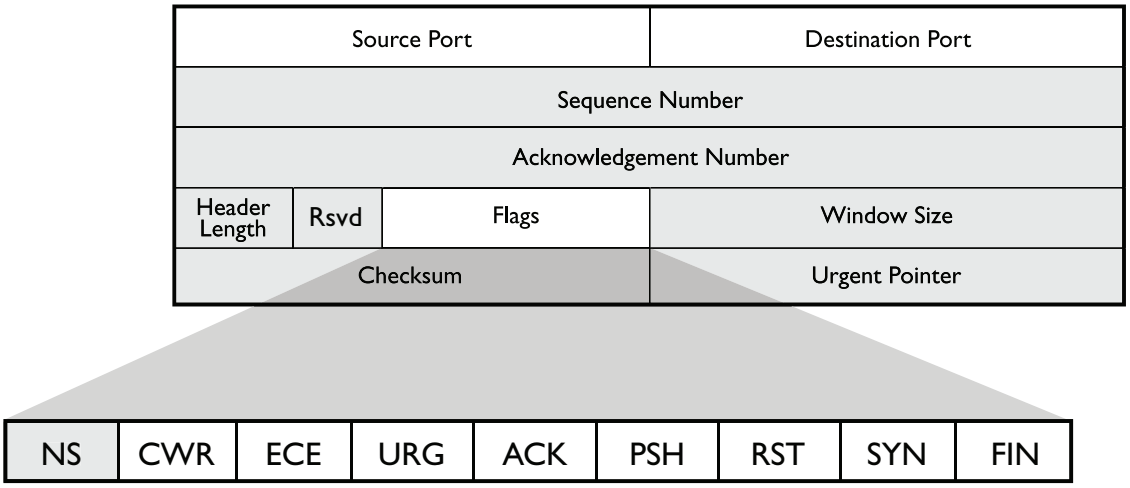

Figure A.2: TCP Header

A.2 Structure of the TCP Header

TCP, the most commonly encountered transport protocol on the Internet, is a stream-based protocol that reliably transmits data from the source to the destination. Figure A.2 shows a breakdown of the TCP header, which adds 20 additional bytes to the IP header. Consequently, TCP packets will always be at least 40 bytes (60 for IPv6) long. As the shaded portions of Figure A.2 show, most of the TCP header information is not retained in SiLK flow records.

A.2.1 TCP Flags

TCP uses *flags* to transmit state information among participants. A flag has two states: high or low; so a flag represents one bit of information. There are six commonly used flags:

ACK: Short for "acknowledge," ACK flags are sent in almost all TCP packets and used to indicate that previously sent packets have been received.

FIN: Short for "finalize," the FIN flag is used to terminate a session. When a packet with the FIN flag is sent, the target of the FIN flag knows to expect no more input data. When both have sent and acknowledged FIN flags, the TCP connection is closed gracefully.

PSH: Short for "push," the PSH flag is used to inform a TCP receiver that the data sent in the packet should immediately be sent to the target application (i.e., the sender has completed this particular send), approximating a message boundary in the stream.

RST: Short for "reset," the RST flag is sent to indicate that a session is incorrect and should be terminated. When a target receives a RST flag, it terminates immediately. Some implementations terminate sessions using RST instead of the more proper FIN sequence.

SYN: Short for "synchronize," the SYN flag is sent at the beginning of a session to establish initial sequence numbers. Each side sends one SYN packet at the beginning of a session.

URG: Short for "urgent" data, the URG flag is used to indicate that urgent data (such as a signal from the sending application) is in the buffer and should be used first. The URG flag should only be seen in Telnet-like protocols such as Secure Shell (SSH). Tricks with URG flags can be used to fool intrusion detection systems (IDS).

Reviewing the state machine will show that most state transitions are handled through the use of SYN, ACK, FIN, and RST. The PSH and URG flags are less directly relevant. Two other rarely used flags are understood by SiLK: ECE (Explicit Congestion Notification Echo) and CWR (Congestion Window Reduced). Neither is relevant to security analysis at this time, although they can be used with the SiLK tool suite if required. A ninth TCP flag, NS (Nonce Sum), is not recognized or supported by SiLK.

A.2.2 Major TCP Services

Traditional TCP services have well-known ports; for example, 80 is Web, 25 is SMTP, and 53 is DNS. IANA maintains a list of these port numbers at https://www.iana.org/assignments/service-names-port-numbers. This list is useful for legitimate services, but it does not necessarily contain new services or accurate port assignments for rapidly changing services such as those implemented via peer-to-peer networks. Furthermore, there is no guarantee that traffic seen (e.g., on port 80) is actually web traffic or that web traffic cannot be sent on other ports.

A.3 Structure of UDP and ICMP Headers

After TCP, the most common protocols on the Internet are UDP and ICMP. Transport layer protocols like TCP and UDP use their port numbers to deliver packets inside the host to the correct process or service. In contrast, IP uses its addressing and routing to deliver packets to the correct interface on the correct host.

While TCP provides other functions, such as data streams and reliability, UDP provides only delivery. UDP does not understand that sequential packets might be related (as in streams); UDP leaves that up to higher layer protocols. UDP does not provide reliability functions, like detecting and recovering lost packets, reordering packets, or eliminating duplicate packets. UDP is a fast but unreliable message-passing mechanism used for services where throughput is more critical than accuracy. Examples include audio/video streaming, as well as heavy-use services such as the Domain Name System (DNS).

ICMP, a reporting protocol that works in tandem with IP, sends error messages and status updates, and provides diagnostic capabilities like echo.

A.3.1 UDP and ICMP Packet Structure

Figure A.3 shows a breakdown of UDP and ICMP packets, as well as the fields collected by SiLK. A UDP packet has both source and destination ports, assigned in the same way TCP assigns them, as well as a payload.

ICMP is a straight message-passing protocol and includes a large amount of information in its first two fields: Type and Code. The Type field is a single byte indicating a general class of message, such as "destination unreachable." The Code field contains a byte indicating greater detail about the type, such as "port unreachable." ICMP messages generally have a limited payload; most messages have a fixed size based on type, with the notable exceptions being echo request (ICMPv4 type 8 or ICMPv6 type 128) and echo reply (ICMPv4 type 0 or ICMPv6 type 129).

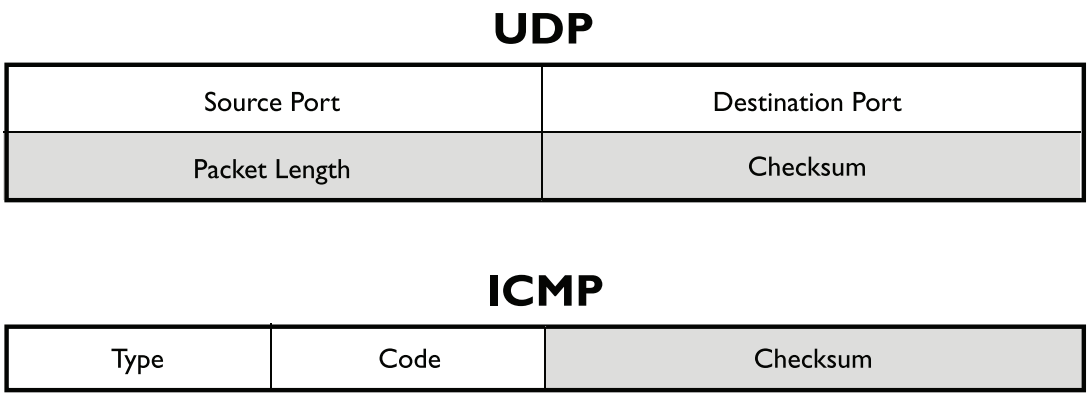

Figure A.3: UDP and ICMP Headers

A.3.2 Major UDP Services and ICMP Messages

UDP services are covered in the IANA webpage whose URL is listed above. As with TCP, the values given by IANA are slightly behind those currently observed on the Internet. IANA also excludes port utilization (even if common) by malicious software such as worms. Although not official, numerous port databases on the web can provide insight into the current port utilization by services.

ICMPv4 types and codes are listed at https://www.iana.org/assignments/icmp-parameters. ICMPv6 types and codes are listed at https://www.iana.org/assignments/icmpv6-parameters. These lists are definitive and include references to RFCs explaining the types and codes.

This page intentionally left blank.

Appendix B

Common Features of SiLK Commands

This appendix lists the features, including parameters, that are common across SiLK commands.

B.1 Getting Help with SiLK Tools

All SiLK tools include a help screen that provides a summary of command information. To view the help screen, specify the `--help` parameter with the command. SiLK is also distributed with UNIX manual (or `man`) pages that explain all the parameters and functionality of each tool in the suite.

```
$ command --help
$ man command
```

B.2 Displaying the Current Version of SiLK

All SiLK tools have a `--version` parameter that identifies the version installed. Since the SiLK tool suite is still being extended and evolved, this version information may be quite important.

```
$ command --version
```

B.3 Parameters Common to Important SiLK Commands

Many of the SiLK tools share features such as common parameters, handling the two versions of IP addresses, and controlling the overwriting of existing output files. Table B.1. shows these options for the most commonly-used SiLK tools.

Table B.2 detailed descriptions of the options in Table B.1. Acceptable values for `--ip-format` are listed and described in Table B.3, values for `--timestamp-format` are described in Table B.4, and values for `--ipv6-policy` are described in Table B.2.

Parameter	rwfilter	rwstats	rwcount	rwcut	rwsort	rwuniq
--help	✓	✓	✓	✓	✓	✓
--legacy-help		✓				
--version	✓	✓	✓	✓	✓	✓
--site-config-file	✓	✓	✓	✓	✓	✓
filenames	✓	✓	✓	✓	✓	✓
--xargs	✓	✓	✓	✓	✓	✓
--print-filenames	✓	✓	✓	✓	✓	✓
--copy-input		✓	✓	✓		✓
--pmap-file	✓	✓		✓	✓	✓
--plugin	✓	✓		✓	✓	✓
--python-file	✓	✓		✓	✓	✓
--output-path		✓	✓	✓	✓	✓
--no-titles		✓	✓	✓		✓
--no-columns		✓	✓	✓		✓
--column-separator		✓	✓	✓		✓
--no-final-delimiter		✓	✓	✓		✓
--delimited		✓	✓	✓		✓
--ipv6-policy		✓		✓		✓
--ip-format		✓		✓		✓
--timestamp-format		✓	✓	✓		✓
--integer-sensors		✓		✓		✓
--integer-tcp-flags		✓		✓		✓
--pager		✓	✓	✓		✓
--note-add	✓				✓	
--note-file-add	✓				✓	
--dry-run	✓			✓		

Table B.1: Common Parameters in Essential SiLK Tools

Parameter	Description
`--help`	Prints usage description and exits
`--legacy-help`	Prints help for legacy switches
`--version`	Prints this program's version and installation parameters
`--site-config-file`	Specifies the name of the SiLK configuration file to use instead of the file in the root directory of the repository
filenames	Specifies one or multiple filenames as non-option arguments
`--xargs`	Specifies the name of a file (or `stdin` if omitted) from which to read input filenames
`--print-filenames`	Displays input filenames on `stderr` as each file is opened
`--copy-input`	Specifies the file or pipe to receive a copy of the input records
`--pmap-file`	Specifies a prefix-map filename and a map name as *mapname*`:`*path* to create a many-to-one mapping of field values to labels. For `rwfilter`, this creates new partitioning options: `--pmap-src-`*mapname*, `--pmap-dst-`*mapname*, and `--pmap-any-`*mapname*. For other tools, it creates new fields `src-`*mapname* and `dst-`*mapname* (see Section 6.2.7)
`--plugin`	For `rwfilter`, creates new switches and partitioning options with a plug-in program written in the C language. For other tools, creates new fields.
`--python-file`	For `rwfilter`, creates new switches and partitioning options with a plug-in program written in Python. For other tools, creates new fields
`--output-path`	Specifies the output file's path
`--no-titles`	Doesn't print column headings
`--no-columns`	Doesn't align neat columns. Deletes leading spaces from each column
`--column-separator`	Specifies the character displayed after each column value
`--no-final-delimiter`	Doesn't display a column separator after the last column
`--delimited`	Combines `--no-columns`, `--no-final-delimiter`, and, if a character is specified, `--column-separator`
`--ipv6-policy`	Determines how IPv4 and IPv6 flows are handled when SiLK has been installed with IPv6 support (see Table B.1)
`--ip-format`	Chooses the format of IP addresses in output (see Table B.3)
`--timestamp-format`	Chooses the format and/or timezone of timestamps in output (see Table B.4)
`--integer-sensors`	Displays sensors as integers, not names
`--integer-tcp-flags`	Displays TCP flags as integers, not strings
`--pager`	Specifies the program used to display output one screen at a time
`--note-add`	Adds a note, specified in this option, to the output file's header
`--note-file-add`	Adds a note from the contents of the specified file to the output file's header
`--dry-run`	Checks parameters for legality without actually processing data

Table B.2: Parameters Common to Several Commands

Value	Description
`canonical`	Displays IPv4 addresses as dotted decimal quad and most IPv6 addresses as colon-separated hexadectets. IPv4-compatible and IPv4-mapped IPv6 addresses will be displayed in a combination of hexadecimal and decimal. For both IPv4 and IPv6, leading zeroes will be suppressed in octets and hexadectets. Double-colon compaction of IPv6 addresses will be performed.
`zero-padded`	Octets are zero-padded to three digits, and hexadectets are zero-padded to four digits. Double-colon compaction is not performed, which simplifies sorting addresses as text.
`decimal`	Displays an IP address as a single, large decimal integer.
`hexadecimal`	Displays an IP address as a single, large hexadecimal integer.
`force-ipv6`	Display all addresses as IPv6 addresses, using only hexadecimal. IPv4 addresses are mapped to the ::FFFF:0:0/96 IPv4-mapped netblock.

Table B.3: `--ip-format` Values

Value	Description
`default`	Formats timestamps as *YYYY/MM/DD*`T`*hh*:*mm*:*ss*.*sss* (`rwcut` and `rwcount` may display milliseconds; `rwuniq` and `rwstats` never do)
`iso`	Formats timestamps as *YYYY-MM-DD hh*:*mm*:*ss*.*sss*
`m/d/y`	Formats timestamps as *MM/DD/YYYY hh*:*mm*:*ss*.*sss*
`epoch`	Formats timestamps as the number of seconds since 1970/01/01 00:00:00 UTC (UNIX epoch) *ssssssssss*.*sss*
`no-msec`	Truncates milliseconds (.*sss*) from `sTime`, `eTime`, and `dur` fields – `rwcut` only
`utc`	Specifies timezone to use Coordinated Universal Time (UTC)
`local`	Specifies timezone to use the `TZ` environment variable or the system timezone

Table B.4: `--timestamp-format` *format*, *modifier*, and *timezone* Values

Appendix C

Additional Information on SiLK

Network Traffic Analysis with SiLK has been designed to provide an overview of data analysis with SiLK on an enterprise network. This overview has included the definition of network flow data, the collection of that data on the enterprise network, and the analysis of that data using the SiLK tool suite. The last chapter provided a discussion on how to extend the SiLK tool suite to support additional analyses.

This handbook provides a large group of analyses in the examples, but these examples are only a small part of the set of analyses that SiLK can support. The SiLK community continues to develop new analytical approaches and provide new insights into how analysis should be done. The authors wish the readers of this handbook good fortune in participation as part of this community.

C.1 SiLK Support and Documentation

The SiLK tool suite is available in open-source form at https://tools.netsa.cert.org/silk.

Before asking others to help with SiLK questions, it is wise to look first for answers in these resources:

***SiLK_tool* `--help`:** All SiLK tools (e.g., `rwfilter` or `rwcut`) support the `--help` option to display terse information about the syntax and usage of the tool.

`man` pages: All SiLK tools have online documentation known as manual pages, or `man` pages, that describe the tool more thoroughly than the `--help` text. The description is not a tutorial, however. `man` pages can be accessed with the `man` command on a system that has SiLK installed or via web links listed on the SiLK Documentation webpage at https://tools.netsa.cert.org/silk/docs.html#manuals.

***The SiLK Reference Guide*:** This guide contains the entire collection of `man` pages for all the SiLK tools in one document. It is provided at https://tools.netsa.cert.org/silk/reference-guide.html in HTML format and at https://tools.netsa.cert.org/silk/reference-guide.pdf in Adobe® Portable Document Format (PDF) .

SiLK Tool Suite Quick Reference booklet: This very compact booklet (located at https://tools.netsa.cert.org/silk/silk-quickref.pdf) describes the dozen most used SiLK commands in a small ($5.5'' \times 8.5''$) format. It also includes tables of flow record fields and transport layer protocols.

SiLK FAQ: This webpage answers frequently asked questions about the SiLK analysis tool suite. Find it at https://tools.netsa.cert.org/silk/faq.html.

C.2 FloCon Conference and Social Media

The CERT Division of the SEI supports FloCon®, an annual international conference devoted to large-scale data analysis for improving the security of networked systems—including flow analysis. More information on FloCon is provided at https://resources.sei.cmu.edu/news-events/events/flocon.

Since the FloCon conference covers a range of network security topics, including network flow analysis, the conference organizers encourage ongoing discussions. In support of this, the following social networking opportunities are offered:

@FloCon_News Twitter account: The FloCon conference organizers post notices related to FloCon here. View (and follow!) the FloCon tweets at https://twitter.com/FloCon_News.

"FloCon Conference" LinkedIn member: The FloCon Conference member page (https://www.linkedin.com/in/flocon) displays postings from the conference organizers, as well as from participants.

"FloCon" LinkedIn group: You can request membership to this private LinkedIn group at https://www.linkedin.com/groups?gid=3636774. Here, members discuss matters related to the FloCon conference and network flow analysis.

C.3 Email Addresses and Mailing Lists

The primary SiLK email addresses and lists are described below:

netsa-tools-discuss@cert.org: This distribution list is for discussion of tools produced by the CERT/CC for network situational awareness, as well as for discussion of flow usage and analytics in general. The discussion might be about interesting usage of the tools or proposals to enhance them. You can subscribe to this list at https://lists.sei.cmu.edu.

netsa-help@cert.org: This email address is for bug reports and general inquiries related to SiLK, especially support with deployment and features of the tools. It provides relatively quick response from CERT/CC users and maintainers of the SiLK tool suite. While a specific response time cannot be guaranteed, this address has proved to be a valuable asset for bugs and usage issues.

netsa-contact@cert.org: This email address provides an avenue for recipients of CERT Situational Awareness Group technical reports to reach the reports' authors. The focus is on analytical techniques. Public reports are provided at https://www.cert.org/netsa/publications.

flocontact@cert.org: General email address for inquiries about FloCon.

flocommunity@cert.org: This distribution list addresses a community of analysts built on the core of the FloCon conference. The list is not focused exclusively on FloCon itself, although it will include announcements of FloCon events.

Appendix D

Further Reading and Resources

This chapter gives helpful background information for many of the topics discussed in this guide. It is intended to be a starting point for further learning!

D.1 Unix and Linux

SiLK is implemented on UNIX (e.g., Apple® OS X®, FreeBSD®, Solaris®) and UNIX-like operating systems and environments (e.g., Linux®, Cygwin). Consequently, an analyst must be able to work with UNIX to use the SiLK tools.

Basic knowledge of UNIX, Linux, or their related operating systems is essential to using SiLK. Many classes, tutorials, and books are available for learning Unix and Linux. The resources in this section can help you to get started.

D.1.1 Useful UNIX Commands

Table D.1 lists some useful UNIX commands. To see more information on these commands type `man` followed by the command name.

D.1.2 Online Tutorials on Unix and Linux

- UNIX Tutorial for Beginners (20+ In-depth Unix Training Videos). *Software Testing Help website.* https://www.softwaretestinghelp.com/unix-tutorials/
- Introduction to Linux (LFS101). *Linux Foundation Website.* https://training.linuxfoundation.org/training/introduction-to-linux/

D.1.3 Books on Unix and Linux

- Peek, Jerry, et al. *Learning the UNIX Operating System (Fifth Edition)*, O'Reilly Media. April, 2014. http://shop.oreilly.com/product/9780596002619.do

Command	Description
`cat`	Copies streams and/or files onto standard output (show file content)
`cd`	Changes [working] directory
`chmod`	Changes file-access permissions. Needed to make script executable
`cp`	Copies a file from one name or directory to another
`cut`	Isolates one or more columns from a file
`date`	Shows the current or calculated day and time
`echo`	Writes arguments to standard output
`exit`	Terminates the current shell or script (log out) with an exit code
`export`	Assigns a value to an environment variable that programs can use
`file`	Identifies the type of content in a file
`grep`	Displays from a file those lines matching a given pattern
`head`	Shows the first few lines of a file's content
`kill`	Terminates a job or process
`less`	Displays a file one full screen at a time
`ls`	Lists files in the current (or specified) directory
	`-l` (for long) parameter to show all directory information
`man`	Shows the online documentation for a command or file
`mkdir`	Makes a directory
`mv`	Renames a file or moves it from one directory to another
`ps`	Displays the current processes
`pwd`	Displays the working directory
`rm`	Removes a file
`sed`	Edits the lines on standard input and writes them to standard output
`sort`	Sorts the contents of a text file into lexicographic order
`tail`	Shows the last few lines of a file
`time`	Shows the execution time of a command
`top`	Shows the running processes with the highest CPU utilization
`uniq`	Reports or omits repeated lines. Optionally counts repetitions
`wc`	Counts the words (or, with `-l` parameter, counts the lines) in a file
`which`	Verifies which copy of a command's executable file is used
`$(...)`	Inserts the output of the contained command into the command line
var=*value*	Assigns a value to a shell variable. For use by the shell only, not programs

Table D.1: Some Common UNIX Commands

- Siever, Ellen, et al. *Linux in a Nutshell: A Desktop Quick Reference (Sixth Edition).* O'Reilly Media. September, 2009. http://shop.oreilly.com/product/9780596154493.do

D.2 Networking and TCP/IP

A knowledge of computer networking and TCP/IP are also essential for effectively using SiLK.

D.2.1 Online Courses on Networking and TCP/IP Fundamentals

- Coursera offers a variety of online courses on networking and TCP/IP. You can get started with *The Bits and Bytes of Networking* (https://www.coursera.org/learn/computer-networking) and Introduction to TCP/IP (https://www.coursera.org/learn/tcpip).

- Udemy Academy also offers online tutorials on networking and TCP/IP. *Introduction to networking for complete beginners* (https://www.udemy.com/course/introduction-to-networking-for-complete-)beginners/ is a good place to start.

D.2.2 Books on Networking and TCP/IP Fundamentals

- Kurose, James, and Ross, Keith. *Computer Networking: A Top-Down Approach.* Pearson, 6th edition. March 5, 2012.

- Lowe, Doug. *Networking All-in-One For Dummies.* 7th edition. For Dummies. April 10, 2018.

- Fall, Kevin R. and Stevens, W. Richard . *TCP/IP Illustrated, Volume 1: The Protocols, Edition 2.* Addison-Wesley. November 8, 2011

D.3 Network Flow and Related Topics

Much has been written on NetFlow, SiLK, and related analysis. Here we provide a non-comprehensive list of examples you may wish to consider. Several related topics will enhance your ability to to use SiLK effectively.

D.3.1 Technical Papers

Rick Hofstede, et al. *Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX*, IEEE Communications Surveys and Tutorials (Volume 16, Issue 4, Fourth quarter 2014) https://ieeexplore.ieee.org/document/6814316/?arnumber=6814316&tag=1

T. Taylor, S. Brooks, J. McHugh. "NetBytes Viewer: An Entity-Based NetFlow Visualization Utility for Identifying Intrusive Behavior." In: Goodall J.R., Conti G., Ma KL. (eds) *VizSEC 2007. Mathematics and Visualization.* Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-78243-8_7#citeas

T. Taylor, D. Paterson, J. Glanfield, C. Gates, S. Brooks and J. McHugh, "FloVis: Flow Visualization System," *2009 Cybersecurity Applications and Technology Conference for Homeland Security*, Washington, DC, 2009, pp. 186-198. doi: 10.1109/CATCH.2009.18 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4804443&isnumber=4804414

Jeff Janies, Red Jack. *Protographs: Graph-Based Approach to NetFlow Analysis.* FloCon 2011 https://resources.sei.cmu.edu/asset_files/Presentation/2011_017_101_50576.pdf

M. Thomas, L. Metcalf, J. Spring, P. Krystosek and K. Prevost, "SiLK: A Tool Suite for Unsampled Network Flow Analysis at Scale," *2014 IEEE International Congress on Big Data*, Anchorage, AK, 2014, pp. 184-191. doi: 10.1109/BigData.Congress.2014.34 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6906777&isnumber=6906742

V. Marinov, J. Schoenwaelder. "Design of an IP Flow Record Query Language." In: D. Hausheer, J. Schoenwaelder (eds) *Resilient Networks and Services.* AIMS 2008. Lecture Notes in Computer Science, vol 5127. Springer, Berlin, Heidelberg, 2008

M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert and C. Kemp,"Detection of SSH Brute Force Attacks Using Aggregated Netflow Data," *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, Miami, FL, 2015, pp. 283-288. doi: 10.1109/ICMLA.2015.20 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7424322&isnumber=7424247

Udaya Wijesinghe, Udaya Tupakula, Vijay Varadharajan. "An Enhanced Model for Network Flow Based Botnet Detection." *Proceedings of the 38th Australasian Computer Science Conference (ACSC 2015)*, Sydney, Australia, 27 - 30 January 2015 http://crpit.com/confpapers/CRPITV159Wijesinghe.pdf

D.3.2 Books on Network Flow and Network Security

Michael W. Lucas. *Network Flow Analysis.* no starch press. June 2010, ISBN-13: 978-1-59327-203-6 https://nostarch.com/networkflow

Omar Santos, *Network Security with NetFlow and IPFIX.* 2016 Cisco Systems, Inc, Cisco Press, Indianapolis, IN http://www.ciscopress.com/store/network-security-with-netflow-and-ipfix-big-data-analytics-9781587144387

D.4 Bash Scripting Resources

Throughout this book, we use bash scripts to organize and execute collections of SiLK commands. There are many books, online classes, and web tutorials on Bash and its uses. Here are some that may be helpful.

D.4.1 Online Tutorial

Online Shell Scripting Tutorial: https://www.shellscript.sh

D.4.2 Books on Bash Scripting

Many books have been written on Bash and shell scripting in general. See the following publishers for their current list of titles relating to Bash.

Google search for Bash scripting books: https://www.google.com/search?q=list+of+books+on+Bash+scripting

O'Reilly
https://www.oreilly.com
https://ssearch.oreilly.com/?q=bash

No Starch Press, Inc.
https://nostarch.com
https://nostarch.com/search/node/bash%20scripting

John Wiley and Sons/WROX
http://www.wrox.com/WileyCDA/Section/id-WROX_SEARCH_RESULT.html?query=bash

Addison Wesley
https://openlibrary.org/search?q=bash+scripting&mode=everything

Linux Training Academy
https://www.linuxtrainingacademy.com/books

D.5 Visualizing SiLK Data

Visualization can be integrated into the workflow for network flow analysis. Graphing the results enables you to find trends and relationships that are hard to spot if you only examine the numeric output of SiLK commands.

Use your favorite spreadsheet, graphing, and numerical analysis tools to generate plots of the intermediate and final results of single path, multi-path, and exploratory analyses. The visualizations in *Network Traffic Analysis with SiLK* were done using Microsoft Excel.

The following document gives an overview of three techniques for visualizing SiLK data (describe, analyze, and explore):

Krystosek, Paul. *Visualization of Network Flow Data.* FloCon 2014. January, 2014. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=300450

D.5.1 Rayon

Rayon is a Python library and set of tools for generating basic, two-dimensional statistical visualizations. Rayon can be used to automate reporting; provide data visualization in command-line, GUI or web applications; or do ad-hoc exploratory data analysis.

Download the Rayon library and documentation from https://tools.netsa.cert.org/rayon/index.html

The following document shows how Rayon visualizations work with the Unix command line and SiLK:

Groce, Phil, *The Rayon Tools: Visualization at the Command Line.* FloCon 2014. January, 2014. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=300462

D.5.2 FloViz

FloViz is a comprehensive and extensible set of visualization tools. It is integrated with the SiLK tool suite via a relational database that stores data such as sets and multisets (bags) that are derived from NetFlow and similar sources.

You can find more information about FloViz at https://web.cs.dal.ca/~sbrooks/projects/NetworkVis/index.html

D.5.3 Graphviz—Graph Visualization Software

Network flow records can be visualized as directed graphs. Graphviz can be used to produce such visualizations. It's open source graph visualization software that represents structural information as diagrams of abstract graphs and networks.

To download Graphviz executables, source code, and documentation, visit https://www.graphviz.org

D.5.4 The Spinning Cube of Potential Doom

First implemented as a demonstration, it is an interesting display of network traffic at the Supercomputing conference in 2004. It has since taken on a life of its own.

http://www.nersc.gov/news-publications/nersc-news/nersc-center-news/1998/cube-of-doom

Stephen Lau. "The Spinning Cube of Potential Doom." *Commun. ACM 47, 6* (June 2004), 25-26. https://dx.doi.org/10.1145/990680.990699

D.6 Networking Standards

Service Name and Transport Protocol Port Number Registry
https://www.iana.org/assignments/service-names-port-numbers

RFC 871, A Perspective on the ARPANET Reference Model.
https://tools.ietf.org/html/rfc871

ETF RFC 3954, Cisco Systems NetFlow Services Export Version 9, 2004
https://www.ietf.org/rfc/rfc3954.txt

RFC 4291, IP Version 6 Addressing Architecture
https://tools.ietf.org/html/rfc4291

RFC 6890, Special-Purpose IP Address Registries
https://tools.ietf.org/html/rfc6890.

IPFIX standards:

RFC3917: Requirements for IP Flow Information Export (IPFIX)

RFC3955: Candidate Protocols for IP Flow Information Export (IPFIX)

RFC5101: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information (IPFIX)

RFC5102: Information Model for IP Flow Information Export

RFC5103: Bidirectional Flow Export Using IP Flow Information Export

RFC5153: IPFIX Implementation Guidelines

RFC5470: Architecture for IP Flow Information Export

RFC5471: Guidelines for IP Flow Information Export (IPFIX) Testing

RFC5472: IP Flow Information Export (IPFIX) Applicability

RFC5473: Reducing Redundancy in IP Flow Information Export (IPFIX) and Packet Sampling (PSAMP) Reports

Index

Carnegie Mellon University
Software Engineering Institute

Network Traffic Analysis with SiLK

Analyst's Handbook for SiLK Version 3.15.0 and Later

August 2020