Using SiLK for Network Traffic Analysis

Analyst's Handbook for SiLK Versions 3.8.3 and Later

Ron Bandes
Timothy Shimeall
Matt Heckathorn
Sidney Faber

October 2014
CERT Coordination Center®

CERT
Software Engineering Institute
Carnegie Mellon University

This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM-0001832

Acknowledgements

The authors wish to acknowledge the valuable contributions of all members of the CERT® Network Situational Awareness 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 following individuals deserve special mention:

- Michael Collins, PhD was responsible for the initial draft of this handbook and for the development of the earliest versions of the SiLK tool suite.
- Mark Thomas, PhD, who transitioned the handbook from Microsoft® Word to LaTeX, patiently and tirelessly answered many technical questions from the authors and shepherded the maturing of the SiLK tool suite.
- Michael Duggan answered frequent questions for the preparation of this handbook, often delving into code and performing experiments to determine the actual working and boundary conditions of SiLK components.
- Andrew Kompanek, who oversaw much of the early transition of SiLK into a more maintainable format, contributed many of the examples in this handbook.
- Marcus Deshon, PhD contributed many examples to this handbook and provided patient guidance to a number of revisions.
- The management of the CERT/CC and the Network Situational Awareness group, in particular Roman Danyliw and Richard Friedberg, have provided consistent guidance and support throughout the evolution of this handbook.

The many users of the SiLK tool suite have also contributed immensely to the evolution of the suite and its tools and are acknowledged gratefully.

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

Contents

List of Figures

List of Tables

List of Examples

Handbook Goals

What This Handbook Covers

This handbook provides a tutorial introduction to network traffic analysis using the System for Internet-Level Knowledge (or SiLK) tool suite. This suite is publicly available at http://tools.netsa.cert.org/silk/ and supports both acquisition and analysis of network flow data. The SiLK tool suite is a highly scalable flow-data capture and analysis system developed by the Network Situational Awareness group (NetSA) at Carnegie Mellon[1] University's Software Engineering Institute (SEI). 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. The SiLK tools provide network security analysts with a relatively complete high-level view of traffic across an enterprise network, subject to placement of sensors.

Analyses Made Possible by SiLK

Analyses using the SiLK tools have lent insight into various aspects of network behavior. Some example applications of this tool suite include (these examples, and others, are explained further in this handbook):

- 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 CIDR[2] 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

By providing a common basis for these various analyses, the tools provide a framework on which network situational awareness may be developed.

[1] Carnegie Mellon is a registered trademark of Carnegie Mellon University.
[2] Classless Inter-Domain Routing

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

- What is on my network?
- What happened before the 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?
- Do I have 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?

How This Handbook Is Organized

This handbook contains six chapters:

1. The **Networking Primer and Review of UNIX® Skills** provides a very brief overview of some of the background necessary to begin using the SiLK tools for analysis. It includes a brief introduction to Transmission Control Protocol/Internet Protocol (TCP/IP) networking and covers some of the UNIX command-line skills required to use the SiLK analysis tools.

2. **The SiLK Flow Repository** describes the structure of network flow data, how they are collected from the enterprise network, and how they are organized.

3. **Essential SiLK Tools** describes how to use the SiLK tools for common tasks including data access, display, simple counting, and statistical description.

4. **Using the Larger SiLK Tool Suite** builds on the previous chapter and covers use of other SiLK tools for data analysis, including manipulating flow record files, analyzing packets, and working with aggregates of flows and IP addresses.

5. **Using PySiLK for Advanced Analysis** discusses how analysts can use the scripting capabilities of PySiLK—the SiLK Python extension—to facilitate more complex analyses efficiently.

6. **Additional Information on SiLK** describes some sources of additional information and assistance that are available for the SiLK tool suite.

What This Handbook Doesn't Cover

This handbook is not an exhaustive description of all the tools in the SiLK tool suite or of all the options in the described tools. Rather, it offers concepts and examples to allow analysts to accomplish needed work while continuing to build their skills and familiarity with the tools. Every tool in the analysis suite accepts a `--help` option that briefly describes the tool. In addition, each tool 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, `man rwfilter` to see information about the `rwfilter` command. The SiLK Documentation page at http://tools.netsa.cert.org/silk/docs.html includes links to individual manual pages. The SiLK Reference Guide is a single document that bundles all the SiLK manual pages. It is available in HTML and PDF formats on the SiLK Documentation page. Various analysis topics are explored via tooltips, available at https://tools.netsa.cert.org/tooltips.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 *SiLK Installation Handbook* (http://tools.netsa.cert.org/silk/install-handbook.pdf).

Chapter 1

Networking Primer and Review of UNIX Skills

This chapter reviews basic topics in Transmission Control Protocol/Internet Protocol (TCP/IP) and UNIX operation. It is not intended as a comprehensive summary of these topics, but it will help to refresh your knowledge and prepare you for using the SiLK tools for analysis.

Upon completion of this chapter you will be able to

- describe the structure of IP packets and the relationship between the protocols that constitute the IP protocol suite
- explain the mechanics of TCP, such as the TCP state machine and TCP flags
- use basic UNIX tools

1.1 Understanding TCP/IP Network Traffic

This section provides an overview of the TCP/IP networking suite. TCP/IP is the foundation of internetworking. All packets analyzed by the SiLK system use protocols supported by the TCP/IP suite. These protocols behave in a well-defined manner, and one possible sign of a security breach can be a deviation from accepted behavior. In this section, you will learn about what is specified as accepted behavior. While there are common deviations from the specified behavior, knowing what is specified forms a basis for further knowledge.

This section is a refresher; the TCP/IP suite is a complex collection of more than 50 protocols, and it comprises far more information than can be covered in this section. A number of online documents and printed books provide other resources on TCP/IP to further your understanding of the TCP/IP suite.

1.1.1 TCP/IP Protocol Layers

Figure 1.1 shows a basic breakdown of the protocol layers in TCP/IP. The Open Systems Interconnection (OSI) Reference Model, the best known model for layered protocols, consists of seven layers. However,

TCP/IP wasn't created with the OSI Reference Model in mind. TCP/IP conforms with the Department of Defense (DoD) Arpanet Reference Model (RFC[3] 871, found at http://tools.ietf.org/html/rfc871), a four-layer model. Although TCP/IP and the DoD Arpanet Reference Model have a shared history, it is useful and customary to describe TCP/IP's functions in terms of the OSI Reference Model. OSI is the only model in which network professionals sometimes refer to the layers by number, so any reference to Layer 4, or L4, definitely refers to OSI's Transport layer.

Figure 1.1: TCP/IP Protocol Layers

OSI Reference Model	**DoD (TCP/IP) Arpanet Ref Model**
7 Application	Process Level / Applications
6 Presentation	
5 Session	
4 Transport	Host-to-Host
3 Network	Internet
2 Data-Link	Network Interface
1 Physical	

Starting with the top row of Figure 1.1, a *network application* (such as email, telephony, streaming television, or file transfer) creates a *message* that should be understandable by another instance of the network application on another host; this is an *application*-layer message. Sometimes the character set, graphics format, or file format must be described to the destination host—as with Multipurpose Internet Mail Extensions (MIME) in email—so the destination host can present the information to the recipient in an understandable way; this is done by adding metadata to the *presentation*-layer header. Sometimes users want to be able to resume communications sessions when their connections are lost, such as with online games or database updates; this is accomplished with the *session*-layer checkpointing capabilities. Many communications do not use functions of the presentation and session layers, so their headers are omitted. The *transport*-layer protocols identify with port numbers which process or service in the destination host should handle the incoming data; a protocol like User Datagram Protocol (UDP) does little else, but a more complicated protocol like TCP also performs packet sequencing, duplicate packet detection, and lost packet retransmission. The *network* layer is where we find Internet Protocol, whose job is to route packets from the network interface of the source host to the network interface of the destination host, across many networks and routers in the internetwork. Those networks are of many types (such as Ethernet, Asynchronous Transfer Mode [ATM], cable modem [DOCSIS®], or digital subscriber line [DSL]), each with its own frame format and rules described by its *data-link*-layer protocol. The data-link protocol imposes a maximum transmission unit (MTU) size on frames and therefore on datagrams and segments as well. The vast majority of enterprise network data is transferred over Ethernet at some point, and Ethernet has the lowest MTU (normally 1,500; 1,492 with IEEE® 802.2 LLC) of any modern Data-Link layer protocol. So Ethernet's MTU becomes the effective MTU for the full path. Finally, the frame's bits are transformed into an energy (electrical, light, or radio wave) signal by the *physical* layer and transmitted across the medium (copper wire, optical fiber, or space). The process of each successively lower layer protocol adding information to the original message is called *encapsulation* because it's like putting envelopes inside other envelopes. Each layer adds metadata to the

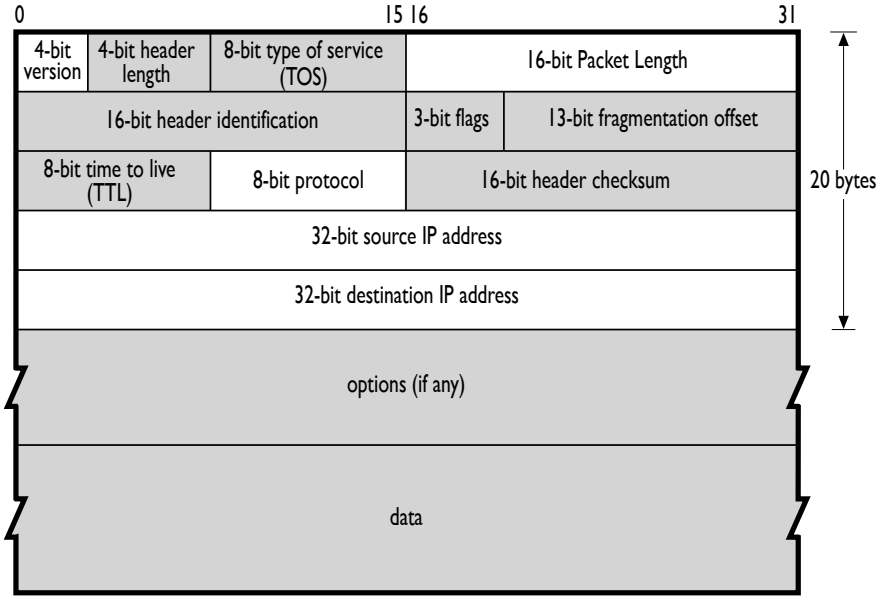
[3] A Request for Comments is an official document, issued by the Internet Engineering Task Force. Some RFCs have Standards status; others do not.

packet that it receives from a higher layer by prepending a header like writing on the outside of that layer's envelope. When a signal arrives at the destination host's network interface, the entire process is reversed with *decapsulation*.

1.1.2 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 1.2 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 1.2: Structure of the IPv4 Header



1.1.3 IP Addressing and Routing

IP can be thought of as a very-high-speed postal service. If someone in Pittsburgh sends a letter to someone in New York, the letter passes through a sequence of postal workers. The postal worker who touches the mail may be different every time a letter is sent, and the only important address is the destination. Normally, there is no reason that New York has to respond to Pittsburgh, and if it does (such as for a return receipt), the sequence of postal workers could be completely different.

IP operates in the same fashion: There is a set of routers between any pair of sites, and packets are sent to the routers the same way that the postal system passes letters back and forth. There is no requirement that the set of routers used to pass data to a destination must be the same as the set used for the return trip, and the routes can change at any time.

Most importantly, the only IP address that must be valid in an IP packet is the destination address. IP itself does not require a valid source address, but some other protocols (e.g., TCP) cannot complete without valid source and destination addresses because the source needs to receive the acknowledgment packets to complete a connection. (However, there are numerous examples of intruders using *incomplete* connections for malicious purposes.)

Structure of an IP Address

The Internet has space for approximately four billion unique IPv4 addresses. While an IPv4 address can be represented as a 32-bit integer, it is usually displayed in *dotted decimal* (or *dotted quad*) format as a set of four decimal integers separated by periods (dots); for example, 128.2.118.3, where each integer is a number from 0 to 255, representing the value of one byte (octet).

IP addresses and ranges of addresses can also be referenced using *CIDR blocks*. CIDR is a standard for grouping together addresses for routing purposes. When an entity purchases or leases a range of IP addresses from the relevant authorities, that entity buys/leases a routing block, that is used to direct packets to its network.

CIDR blocks are usually described with CIDR notation, consisting of an address, a slash, and a prefix length. The prefix length is an integer denoting the number of bits on the left side of the address needed to identify the block. The remaining bits are used to identify hosts within the block. For example, 128.2.0.0/16 would signify that the leftmost 16 bits (2 octets), whose value is 128.2, identify the CIDR block and the remaining bits on the right can have any value denoting a specific host within the block. So all IP addresses from 128.2.0.0 to 128.2.255.255, in which the first 16 bits are unchanged, belong to the same block. Prefix lengths range from 0 (all addresses belong to the same unspecified network; there are 0 network bits specified)[4] to 32 (the whole address is made of unchanging bits, so there is only one address in the block; the address is a single host).

With the introduction of IPv6, all of this is changing. IPv6 addresses are 128 bits in length, for a staggering 3.4×10^{38} (340 undecillion or 340 trillion trillion trillion) possible addresses. IPv6 addresses are represented as groups of eight hexadectets (four hexadecimal digit integers); for example

```
FEDC:BA98:7654:3210:0037:6698:0000:0510
```

Each integer is a number between 0 and FFFF (the hexadecimal equivalent of decimal 65,535). IPv6 addresses are allocated in a fashion such that the high-order and low-order digits are manipulated most often, with long strings of hexadecimal zeroes in the middle. There is a shorthand of `::` that can be used once in each address to represent a series of zero groups. The address `FEDC::3210` is therefore equivalent to `FEDC:0:0:0:0:0:0:3210`.

IPv4-compatible (`::0:0/96`) and IPv4-mapped (`::FFFF:0:0/96`) IPv6 addresses are displayed by the SiLK tools in a mixed IPv6/IPv4 format (complying with the canonical format), with the network prefix displayed in hexadecimal, and the 32-bit field containing the embedded IPv4 address displayed in dotted quad decimal. For example, the IPv6 addresses `::102:304` (IPv4-compatible) and `::FFFF:506:708` (IPv4-mapped) will be displayed as `::1.2.3.4` and `::FFFF:5.6.7.8`, respectively.

The routing methods for IPv6 addresses are beyond the scope of this handbook—see RFC 4291 (http://tools.ietf.org/html/rfc4291) for a description. Blocks of IPv6 addresses are generally denoted with CIDR notation, just as blocks of IPv4 addresses are. CIDR prefix lengths can range from 0 to 128 in IPv6. For

[4] CIDR /0 addresses are used almost exclusively for empty routing tables and are not accepted by the SiLK tools. This effectively means the range for CIDR prefix lengths is 1–32 for IPv4.

Table 1.1: IPv4 Reserved Addresses

Space	Description	RFC
0.0.0.0/8	"This host on this network"	1122
10.0.0.0/8	Private networks	1918
100.64.0.0/10	Carrier-grade Network Address Translation	6598
127.0.0.0/8	Loopback (self-address)	6890
169.254.0.0/16	Link local (autoconfiguration)	6890
172.16.0.0/12	Private networks	1918
192.0.0.0/24	Reserved for IETF protocol assignments	6890
192.0.0.0/29	Dual-Stack Lite	6333
192.0.2.0/24	Documentation (example.com or example.net)	5737
192.88.99.0/24	6to4 relay anycast (border between IPv6 and IPv4)	3068
192.168.0.0/16	Private networks	1918
198.18.0.0/15	Network Interconnect Device Benchmark Testing	2544
198.51.100.0/24	Documentation (example.com or example.net)	5737
203.0.113.0/24	Documentation (example.com or example.net)	5737
224.0.0.0/4	Multicast	5771
240.0.0.0/4	Future use (except 255.255.255.255)	1112
255.255.255.255	Limited broadcast	919

example, `::FFFF:0:0/96` indicates that the most significant 96 bits of the address `::FFFF:0:0` constitute the network prefix (or network address), and the remaining 32 bits constitute the host part.

In SiLK, the support for IPv6 is controlled by configuration. Check for IPv6 support by running *any_SiLK _tool* `--version` (e.g., `rwcut --version`). Then examine the output to see if "IPv6 flow record support" is "yes."

Reserved IP Addresses

While IPv4 has approximately four billion addresses available, large segments of IP address space are reserved for the maintenance and upkeep of the Internet. Various authoritative sources provide lists of the segments of IP address space that are reserved. One notable reservation list is maintained by the Internet Assigned Numbers Authority (IANA) at http://www.iana.org/assignments/ipv4-address-space. IANA also keeps a list of IPv6 reservations at http://www.iana.org/assignments/ipv6-address-space.

In addition to this list, the Internet Engineering Task Force (IETF) maintains several RFCs that specify other reserved spaces. Most of these spaces are listed in RFC 6890, "Special-Purpose IP Address Registries" at http://tools.ietf.org/html/rfc6890. Table 1.1 summarizes major IPv4 reserved spaces. IPv6 reserved spaces are shown in Table 1.2.

Examples in this handbook use addresses in the private and documentation spaces, or addresses that are obviously fictitious, such as 1.2.3.4. This is done to protect the identities of organizations on whose data we tested our examples. Analysts may observe, in real captured traffic, addresses that are not supposed to appear on the Internet. This may be due to misconfiguration of network infrastructure devices or to falsified (spoofed) addressing.

Table 1.2: IPv6 Reserved Addresses

Space	Description	RFC
::	"Unspecified" address (source) and default unicast route address (destination) [similar to 0.0.0.0]	4291
::1	Loopback address [similar to 127.0.0.0/8]	4291
::0.0.0.0/96	IPv4-compatible addresses (deprecated by RFC 4291)	1933
::FFFF:0.0.0.0/96	IPv4-mapped addresses	4291
64:FF9B::0.0.0.0/96	IPv4-IPv6 translation with well-known prefix	6052
100::/64	Discard-only address block	6666
2001::/23	IETF protocol assignments	2928
2001::/32	Teredo tunneling	4380
2001:2::/48	Benchmarking	5180
2001:10::/28	Overlay Routable Cryptographic Hash IDentifiers (ORCHID)	4843
2001:DB8::/32	Documentation addresses [similar to 192.0.2.0/24]	3849
2002::/16	6to4 addresses [related to 192.88.99.0/24]	3056
FC00::/7	Unique local addresses [similar to RFC 1918 private addresses] primarily seen as FD00::/8	4193
FE80::/10	Link-local unicast (similar to 169.254.0.0/16)	4291
FEC0::/10	Formerly reserved for site-local unicast addresses (deprecated by RFC 3879)	1884
FF00::/8	Multicast [similar to 224.0.0.0/4]	4291

In general, link-local (169.254.0.0/16 in IPv4, FE80::/10 in IPv6) and loopback (127.0.0.0/8 and ::1) destination IP addresses should not cross any routers. Private IP address space (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, and FC00::/7) should not enter or traverse the Internet, so it should not appear at edge routers. Consequently, the appearance of these addresses at these routers indicates a failure of routing policy. Similarly, traffic should not come into the enterprise network from these addresses; the Internet as a whole should not route that traffic to the enterprise network.

1.1.4 Major Protocols

Protocol Layers and Encapsulation

In the multi-layered scheme used by TCP/IP, lower layer protocols *encapsulate* higher layer protocols, like envelopes within envelopes. When we open the innermost envelope, we find the message that belongs to the highest layer protocol. Conceptually, the envelopes have metadata written on them. In practice, the metadata are recorded in *headers*. The header for the lowest layer protocol is sent over the network first, followed by the headers for progressively higher layers. Finally, the message from the highest layer protocol is sent after the last header.

TCP/IP was created before the OSI Reference Model. But if we refer to a layer by its number (e.g., Layer 3 or L3), we always mean the specified layer in that model. While the preceding description of encapsulation is generally true, the model actually assigns protocols to layers based on the protocol's functions, not its order of encapsulation. This is most apparent with Internet Control Message Protocol (ICMP), which the model assigns to the Network layer (L3), even though its header and payload are encapsulated by IP, which is also

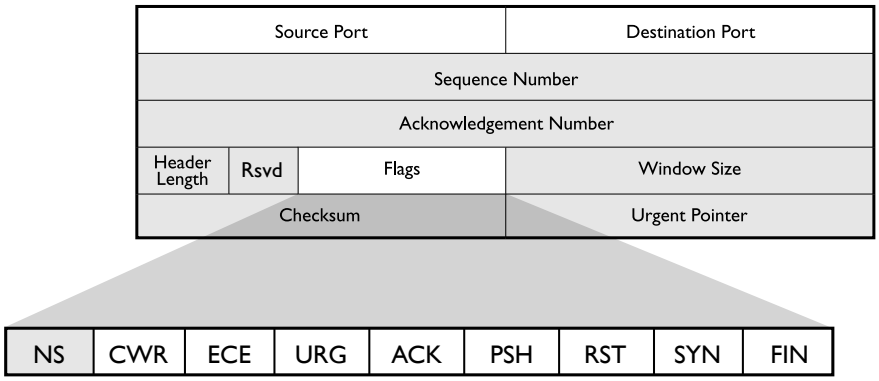
a Network layer protocol. From here on, we will ignore this fine distinction, and we will consider ICMP to be a Transport layer (L4) protocol because it is encapsulated by IP, a Layer 3 protocol.

Transmission Control Protocol (TCP)

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. To maintain this reliability, TCP is very complex: The protocol is slow and requires a large commitment of resources.

Figure 1.3 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 1.3 show, most of the TCP header information is not retained in SiLK flow records.

Figure 1.3: TCP Header

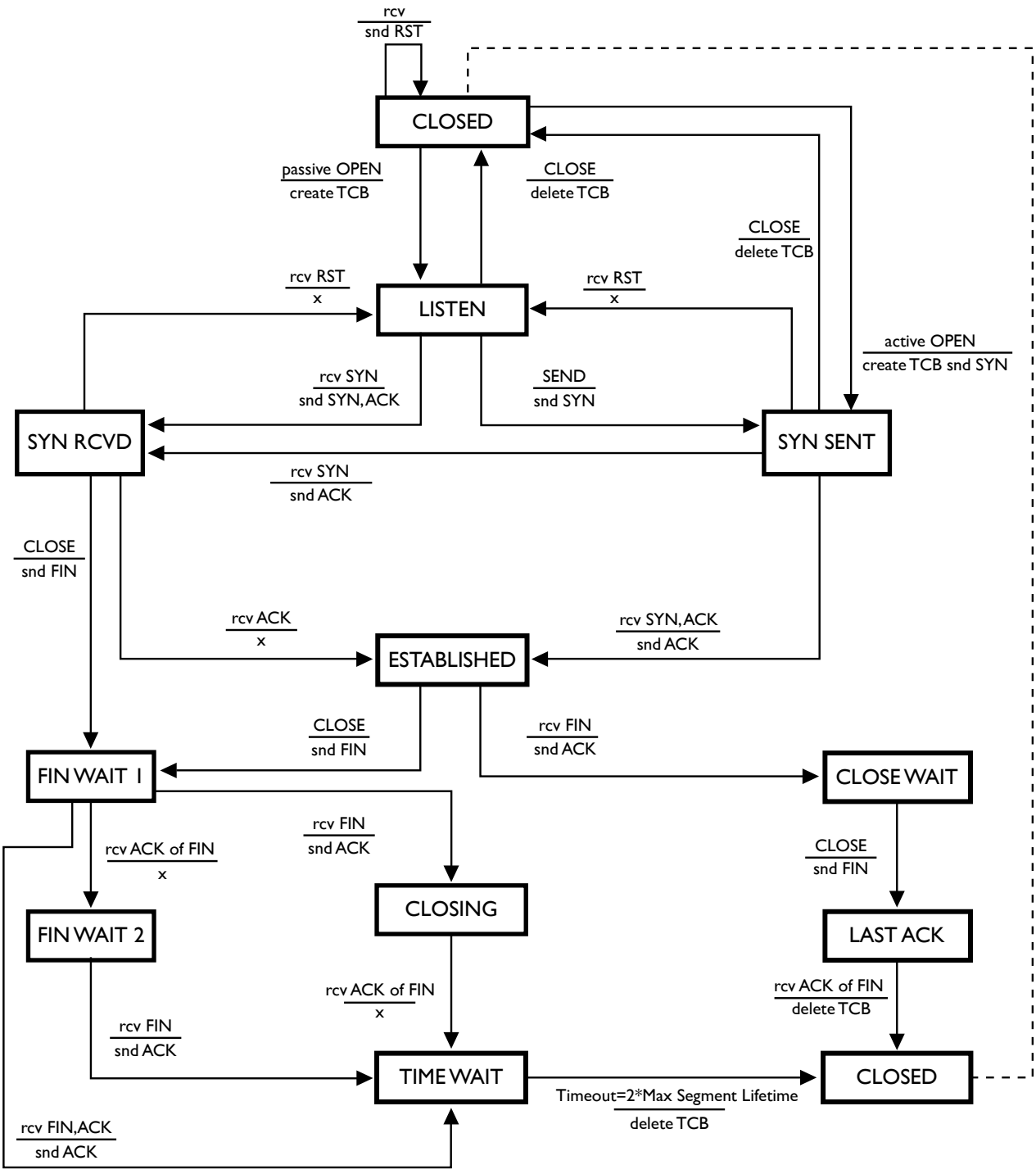


TCP is built on top of an unreliable infrastructure provided by IP. IP assumes that packets can be lost without a problem, and that responsibility for managing packet loss is incumbent on services at higher layers. TCP, which provides ordered and reliable streams on top of this unreliable packet-passing model, implements this feature through a complex state machine as shown in Figure 1.4. The transitions in this state machine are described by labels in a $\frac{stimulus}{action}$ format, where the top value is the stimulating event and the bottom values are actions taken prior to entry into the destination state. Where no action takes place, an "x" is used to indicate explicit inaction.

This handbook does not thoroughly describe the state machine in Figure 1.4 (see http://tools.ietf.org/html/rfc793 for a complete description), however, flows representing well-behaved TCP sessions will behave in certain ways. For example, a flow for a complete TCP session must have at least four packets: one packet that sets up the connection, one packet that contains the data, one packet that terminates the session, and one packet acknowledging the other side's termination of the session.[5] TCP behavior that deviates from this provides indicators that can be used by an analyst. An intruder may send packets with odd TCP flag combinations as part of a scan (e.g., with all flags set on). Different operating systems handle protocol violations differently, so odd packets can be used to elicit information that identifies the operating system in use or to pass through some systems benignly, while causing mischief in others.

[5]It is technically possible for there to be a valid three-packet complete TCP flow: one SYN packet, one SYN-ACK packet containing the data, and one RST packet terminating the flow. This is a very rare circumstance; most complete TCP flows have more than four packets.

Figure 1.4: TCP State Machine



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).

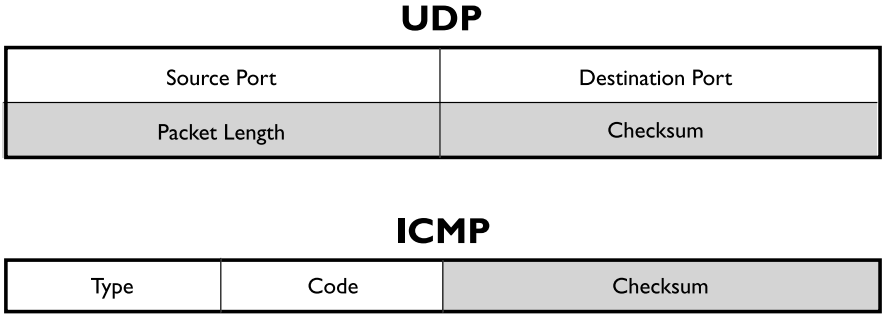
Reviewing the state machine will show that most state transitions are handled through the use of SYN, ACK, FIN, and RST. The PSH and URG flags are less directly relevant. Two other rarely used flags are understood by SiLK: ECE (Explicit Congestion Notification Echo) and CWR (Congestion Window Reduced). 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.

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 http://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.

UDP and ICMP

After TCP, the most common protocols on the Internet are UDP and ICMP. While IP uses its addressing and routing to deliver packets to the correct interface on the correct host, Transport layer protocols like TCP and UDP use their port numbers to deliver packets inside the host to the correct process or service. Whereas TCP also 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.

Figure 1.5: UDP and ICMP Headers



UDP and ICMP Packet Structure

Figure 1.5 shows a breakdown of UDP and ICMP packets, as well as the fields collected by SiLK. UDP can be thought of as TCP without the additional state mechanisms; 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).

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 http://www.iana.org/assignments/icmp-parameters. ICMPv6 types and codes are listed at http://www.iana.org/assignments/icmpv6-parameters. These lists are definitive and include references to RFCs explaining the types and codes.

1.2 Using UNIX to Implement Network Traffic Analysis

This section provides a review of basic UNIX operations. 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.

1.2.1 Using the UNIX Command Line

UNIX uses a program known as a shell to obtain commands from a user and either perform the task described by that command or invoke another program that will. Linux usually uses Bash (Bourne-Again SHell) for its shell. When the shell is ready to accept a command from the user, it displays a string of characters known as a prompt to let the user know that he or she can enter a command now. Besides notifying the user that a command can be accepted at this time, the prompt may convey additional information. The choice of information to be conveyed may be made by the user by providing a prompt template to the shell. In this handbook, the prompt will appear as in Example 1.1.

Example 1.1: A UNIX Command Prompt

```
<1>$
```

The integer between angle brackets will be used to refer to specific commands in examples. Commands can be invoked by typing them directly at the command line. UNIX commands are typically abbreviated English words and accept space-separated parameters. Parameters are just values (like filenames), an option-name/value pair, or just an option name. Option names are double dashes followed by hyphenated words, single dashes followed by single letters, or (rarely) single dashes followed by words (as in the `find` command). Table 1.3 lists some of the more common UNIX commands. To see more information on these commands type `man` followed by the command name. Example 1.2 and the rest of the examples in this handbook show the use of some of these commands.

1.2.2 Standard In, Out, and Error

Many UNIX programs, including most of the SiLK tools, have a default for where to obtain input and where to write output. The symbolic filenames `stdin`, `stdout`, and `stderr` are not the names of disk files, but rather they indirectly refer to files. Initially, the shell assigns the keyboard to stdin and assigns the screen to stdout and stderr. Programs that were written to read and write through these symbolic filenames will default to reading from the keyboard and writing to the screen. But the symbolic filenames can be made to refer indirectly to other files, such as disk files, through shell features called *redirection* and *pipes*.

Output Redirection

Some programs, like `cat` and `cut`, have no way for the user to tell the program *directly* which file to use for output. Instead these programs always write their output to stdout. The user must inform *UNIX*, not the program, that stdout should refer to the desired file. The program then only knows its output is going to stdout, and it's up to UNIX to route the output to the desired file. One effect of this is that any error message emitted by the program that refers to its output file can only display "stdout," since the actual output filename is unknown to the program.

The shell makes it easy to tell UNIX that you wish to *redirect* stdout from its default (the screen) to the file that the user specifies. This is done right on the same command line that runs the program, using the greater than symbol (`>`) and the desired filename (as shown in Command 1 of Example 1.3).

SiLK tools that write binary (non-text) data to stdout will emit an error message and terminate if stdout is assigned to a terminal device. Such tools must have their output directed to a disk file or piped to a SiLK tool that reads that type of binary input.

Table 1.3: Some Common UNIX Commands

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

Example 1.2: Using Simple UNIX Commands

```
<1>$ echo Here are some simple commands:
Here are some simple commands:
<2>$ date
Thu Jul  3 15:56:24 EDT 2014
<3>$ date -u
Thu Jul  3 19:56:24 UTC 2014
<4>$ # This is a comment line. It has no effect.
<5>$ #The next command lists my running processes
<6>$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
user1     8280  8279  0 14:43 pts/2    00:00:00 -bash
user1    10358 10355  1 15:56 pts/2    00:00:00 ps -f
<7>$ cat animals.txt
Animal  Legs    Color
------  ----    -----
fox     4       red
gorilla 2       silver
spider  8       black
moth    6       white
<8>$ file animals.txt
animals.txt: ASCII text
<9>$ head -n 3 animals.txt
Animal  Legs    Color
------  ----    -----
fox     4       red
<10>$ cut -f 1,3 animals.txt
Animal  Color
------  -----
fox     red
gorilla silver
spider  black
moth    white
```

Example 1.3: Output Redirection

```
<1>$ cut -f 1,3 animals.txt >animalcolors.txt
<2>$ cat animalcolors.txt
Animal  Color
------  -----
fox     red
gorilla silver
spider  black
moth    white
<3>$ rm animalcolors.txt
<4>$ ls animalcolors.txt
ls: animalcolors.txt: No such file or directory
```

Input Redirection

A very few programs, like `tr`, have no syntax for specifying the input file and rely entirely on UNIX to connect an input file to stdin. The shell provides a method for redirecting input very similar to redirecting output. You specify a less than symbol (`<`) followed by the input filename as shown in Command 2 of Example 1.4.

Example 1.4: Input Redirection

```
<1>$ #Translate hyphens to slashes
<2>$ tr - / <animals.txt
Animal  Legs    Color
//////  ////    /////
fox     4       red
gorilla 2       silver
spider  8       black
moth    6       white
```

Pipes

The real power of stdin and stdout becomes apparent with *pipes*. A pipe connects the stdout of the first program to the stdin of a second program. This is specified in the shell using a vertical bar character (`|`), known in UNIX as the pipe symbol.

Example 1.5: Using a Pipe

```
<1>$ head -n 4 animals.txt | cut -f 1,3
Animal  Color
------  -----
fox     red
gorilla silver
```

In Example 1.5, the `head` program read the first four lines from the animals.txt file and wrote those lines to stdout as normal, except that stdout does not refer to the screen. The `cut` program has no input filename specified and was programmed to read from stdin when no input filename appears on the command line. The pipe connects the stdout of `head` to the stdin of `cut` so that `head`'s output lines become `cut`'s input lines without those lines ever touching a disk file. `cut`'s stdout was not redirected, so its output appears on the screen.

Here-Documents

Sometimes we have a small set of data that is manually edited and perhaps doesn't change from one run of a script to the next. If so, instead of creating a separate data file for the input, we can put the input data right into the script file. This is called a *here-document*, because the data are right here in the script file, immediately following the command that reads them.

Example 1.6 illustrates the use of a here-document to supply several filenames to a SiLK program called `rwsort`. The `rwsort` program has an option called `--xargs` telling it to get a list of input files from `stdin`. The here-document supplies data to `stdin` and is specified with double less than symbols (`<<`), followed by a string that defines the marker that will indicate the end of the here-document data. The lines of the script file that follow the command are input data lines until a line with the marker string is reached.

Example 1.6: Using a Here-Document

```
<1>$ rwsort --xargs --fields=sTime --output-path=week.rw <<END-OF-LIST
sunday.rw
monday.rw
tuesday.rw
wednesday.rw
thursday.rw
friday.rw
saturday.rw
END-OF-LIST
<2>$ rwfileinfo --fields=count-records *day.rw week.rw
```

Named Pipes

Using the pipe symbol, a script creates an *unnamed* pipe. Only one unnamed pipe can exist for output from a program, and only one can exist for input to a program. For there to be more than one, you need some way to distinguish one from another. The solution is *named* pipes.

Unlike unnamed pipes, which are created in the same command line that uses them, named pipes must be created prior to the command line that employs them. As named pipes are also known as *FIFOs* (for First In First Out), the command to create one is `mkfifo` (make FIFO). Once the FIFO is created, it can be opened by one process for reading and by another process (or multiple processes) for writing.

Scripts that use named pipes often employ another useful feature of the shell: running programs in the background. In Bash, this is specified by appending an ampersand (`&`) to the command line. When a program runs in the background, the shell will not wait for its completion before giving you a command prompt. This allows you to issue another command to run concurrently with the background program. You can force a script to wait for the completion of background programs before proceeding by using the `wait` command.

SiLK applications can communicate via named pipes. In Example 1.7, we create a named pipe (in Command 1) that one call to `rwfilter` (in Command 2) uses to filter data concurrently with another call to `rwfilter` (in Command 3). Results of these calls are shown in Commands 5 and 6. Using named pipes, sophisticated SiLK operations can be built in parallel. A backslash at the very end of a line indicates that the command is continued on the following physical line.

Example 1.7: Using a Named Pipe

```
<1>$ mkfifo /tmp/namedpipe1
<2>$ rwfilter --start-date=2014/03/21T17 --end-date=2014/03/21T18 \
              --type=all --protocol=6 \
              --fail=/tmp/namedpipe1 --pass=stdout \
      | rwuniq --fields=protocol --output-path=tcp.out &
<3>$ rwfilter /tmp/namedpipe1 --protocol=17 --pass=stdout \
      | rwuniq --fields=protocol --output-path=udp.out &
<4>$ wait
<5>$ cat tcp.out
pro|   Records|
  6|  34866860|
<6>$ cat udp.out
pro|   Records|
 17|  17427015|
<7>$ rm /tmp/namedpipe1 tcp.out udp.out
```

1.2.3 Script Control Structures

Some advanced examples in this handbook will use control structures available from Bash. The syntax

`for` *name* `in` *word-list-expression*`; do ... done`

indicates a loop where each of the space-separated values returned by *word-list-expression* is given in turn to the variable indicated by *name* (and referenced in commands as $*name*), and the commands between `do` and `done` are executed with that value. The syntax

`while` *expression*`; do ... done`

indicates a loop where the commands between `do` and `done` are executed as long as *expression* evaluates to true.

Chapter 2

The SiLK Flow Repository

This chapter introduces the tools and techniques used to store information about sequences of packets as they are collected on an enterprise network for SiLK (referred to as "network flow" or "network flow data" and occasionally just "flow"). This chapter will help an analyst become familiar with the structure of network flow data, how the collection system gathers network flow data from sensors, and how to access 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
- use the `rwsiteinfo` command to organize and display information from the site configuration file

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 data are collected to support several different types of analyses of network traffic (some of which are described later in this handbook).

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 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.

A network flow often covers multiple packets, which are grouped together under 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.

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 but is determined by several configuration parameters), and even at that size you may collect many gigabytes of flow records daily on a busy network.

2.1.1 Structure of a Flow Record

A flow file is a series of flow records. A flow record holds all the data SiLK retains from the collection process: the flow label fields, start time, number of packets, duration of flow, and so on. All the fields in the flow record are listed in Table 3.11 on page 59.

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.

2.2 Flow Generation and Collection

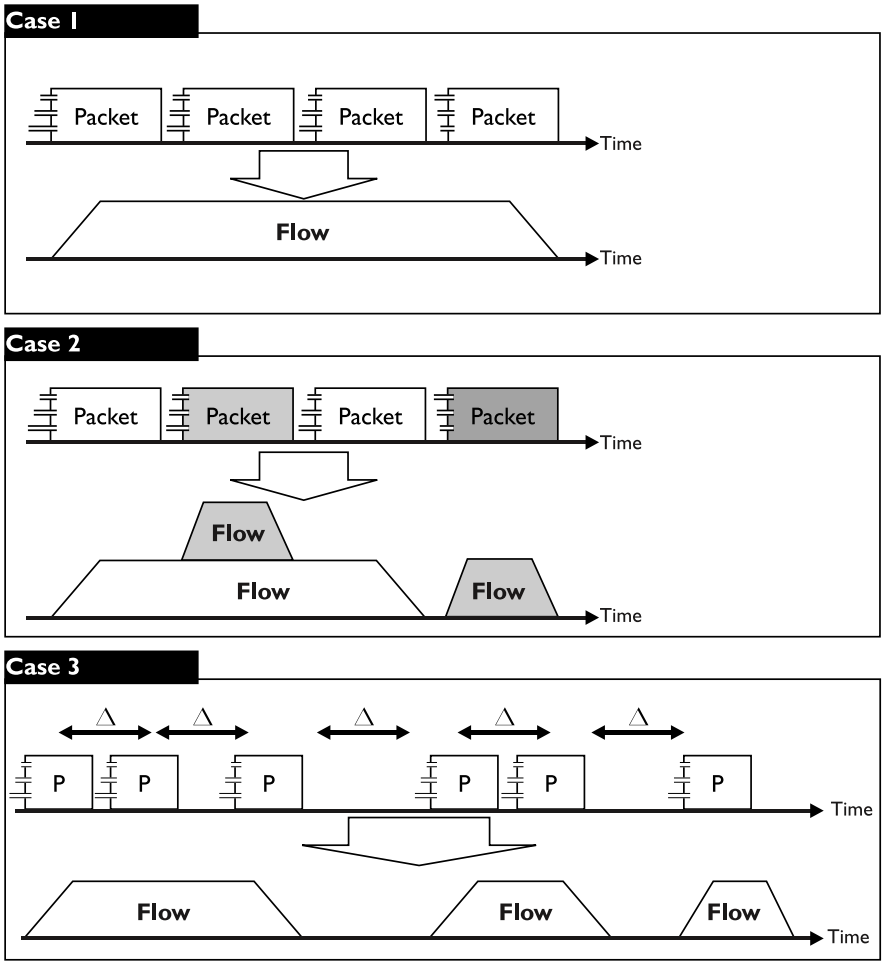
Every day, SiLK may collect many gigabytes of network flow data from across the enterprise network. Given both the volume and complexity of these data, it is critical to understand how these data are recorded. This section reviews the collection process and shows how data are stored as network flow records.

A network flow record is generated by sensors throughout the enterprise network. The majority of these may be routers, although specialized sensors, such as `yaf` (http://tools.netsa.cert.org/yaf/), also can be used to avoid artifacts in a router's implementation of network flow or to use non-device-specific network flow data formats, such as IPFIX (see http://tools.ietf.org/html/rfc7011 for definitions and the IPFIX protocol description and http://www.iana.org/assignments/ipfix for descriptions of the IPFIX information elements), or for more control over network flow record generation.[6] `yaf` also is useful when a data feed from a router is

[6] `yaf` also may be used to convert packet data to network flow records via a script that automates this process. See Section 4.2.

not available, such as on a home network or on an individual host. 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 2.1 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.

Figure 2.1: From Packets to Flows



Network flow is an approximation of traffic, not a natural law. 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:

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. The router has limited processing and memory resources devoted to network flow. During periods of stress, the flow cache will fill and empty more often 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.

2.3 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 majority of this document describes how to use that 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.

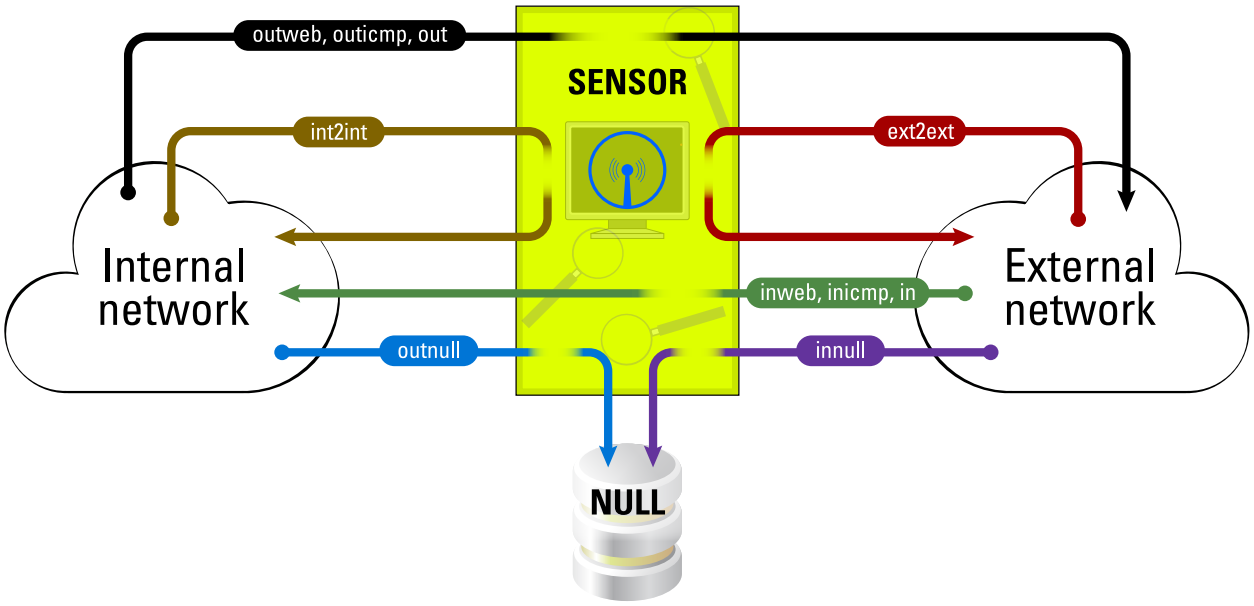
2.3.1 Where Network Flow Data Are Collected

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 2.2. The default implementation has only one class of sensors: *all*. Further segregation of the data is done by type of traffic.

The SiLK tool `rwsiteinfo` can produce a list of sensors in use for a specific installation, reflecting its configuration. It replaces the deprecated[7] `mapsid` command, which only displays sensor information. The `rwsiteinfo` command can display all information from the site configuration. Example 2.1 shows calls to `rwsiteinfo`. Options must be specified with `rwsiteinfo`; there is no default output. Looking at Example 2.1, Command 1 shows how `rwsiteinfo` produces a list of all sensors, displaying both the sensor's numerical ID and its name. Command 2 adds to each sensor the list of classes to which the sensor belongs. Command 3 lists information only for selected sensors, with the sensors specified by their numerical IDs; the descriptive text from the site configuration file (normally silk.conf in the root of the repository) also is requested. Command 4 also lists information only for specified sensors but identifies the sensors by their names. Additionally, the order of the columns is changed with the `--fields` option to keep the specified names in the leftmost column. Command 5 turns its attention away from sensors and displays the SiLK types associated with each class.

[7] A deprecated command is one that has been superseded by a newer command or feature. Deprecated commands are still available for use but will be made obsolete at some time. You should move to the newer command/feature.

Figure 2.2: Default Traffic Types for Sensors



Example 2.1: Using `rwsiteinfo` to Obtain a List of Sensors

```
<1>$ rwsiteinfo --fields=id-sensor,sensor
Sensor-ID|    Sensor|
        0|  SEN-CENT|
        1| SEN-NORTH|
        2| SEN-SOUTH|
        3|  SEN-EAST|
        4|  SEN-WEST|
<2>$ rwsiteinfo --fields=id-sensor,sensor,class:list | head -n 4
Sensor-ID|    Sensor|Class:list|
        0|  SEN-CENT|      c1,c2|
        1| SEN-NORTH|   c1,c2,c3|
        2| SEN-SOUTH|      c1,c2|
<3>$ rwsiteinfo --fields=id-sensor,sensor,describe-sensor --sensors=0,2,4
Sensor-ID|    Sensor|Sensor-Description|
        0|  SEN-CENT|    Central Elbonia|
        2| SEN-SOUTH|   Southern Elbonia|
        4|  SEN-WEST|    Western Elbonia|
<4>$ rwsiteinfo --fields=sensor,id-sensor --sensors=SEN-NORTH,SEN-EAST
   Sensor|Sensor-ID|
SEN-NORTH|         1|
 SEN-EAST|         3|
<5>$ rwsiteinfo --fields=class,type:list
Class|                          Type:list|
   c1|in,out,inweb,outweb,int2int,ext2ext|
   c2|in,out,inweb,outweb,int2int,ext2ext|
   c3|in,out,inweb,outweb,int2int,ext2ext|
```

2.3.2 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 2.2), 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 out 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 (usually indicates a configuration problem either in the sensor or at the ISP)

The additional types are

inicmp and *outicmp* (operational only if SiLK was compiled with the option to support these types), which represent ICMP traffic entering or leaving the enterprise network

innull and *outnull*, which only can be found when the sensor is a router and not a dedicated sensor, and which 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 in which one of the addresses (source or destination) is in neither the internal nor the external networks

These types are configurable, and configurations vary as to which types are in actual use (see the discussion below on sensor class and type). There is also a constructed type *all* that selects all types of flows associated with a class of sensors.

2.3.3 The Collection System and Data Management

To understand how to use SiLK for analysis, it helps to have some understanding of how 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.

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 (depending on timeout configuration and on sensor caching, among other factors) after the start time of the first packet in the flow.

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, 2014 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/2014/03/01/in-SEN1_20140301.15`.

Important Considerations when Accessing Flow Data

While SiLK allows rapid access and analysis of network traffic data, the amount of data crossing the enterprise network could be extremely large. A variety of techniques can optimize the queries, and this section reviews some general guidelines for more rapid data analysis.

Usually, the amount of data associated with any particular event is relatively small. All the traffic from a particular workstation or server may be recorded in a few thousand records at most for a given day. Most of the time, an initial query involves simply pulling and analyzing the relevant records.

As a result, query time can be reduced by simply manipulating the selection parameters, in particular `--type`, `--start-date`, `--end-date`, and `--sensors`. If you know when a particular event occurred, reducing the search time frame by using `--start-date` and `--end-date`'s hour facilities will increase efficiency (e.g., `--start-date=2014/3/1T12 --end-date=2014/3/1T14` is more efficient than `--start-date=2014/3/1T00 --end-date=2014/3/1T23`).

Another useful, but less certain, technique is to limit queries by sensor. Since routing is relatively static, the same IP address will generally enter or leave through the same sensor, which can be determined by using `rwuniq --fields=sensor` (see Section 3.7) and a short (one-hour) probe on the data to identify which sensors are associated with a particular IP address. This technique is especially applicable for long (such as multi-month) queries and usually requires some interaction, since rerouting does occur during normal operation. To use this technique for long queries, start by identifying the sensors using `rwuniq`, query for some extensive period of time using those sensors, and then plot the results using `rwcount`. If analysts see a sudden drop in traffic from those sensors, they should check the data around the time of this drop to see if traffic was routed through a different sensor.

2.3.4 How Network Flow Data Are Organized

The data repository is accessed through the use of SiLK tools, particularly the `rwfilter` command-line application. An analyst using `rwfilter` should specify the type of data to be viewed by using a set of five selection parameters. This handbook discusses selection parameters in more depth in Section 3.2; this section briefly outlines how data are stored in the repository.

Dates

Repository data are stored 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, 2014 in Pittsburgh would be referred to as `2014/2/23T16` 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 2005/3/7 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 2.2, each class of sensor has several types of traffic associated with it: typically in, inweb, out, and outweb. To find the classes and types supported by the 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.

Data types are used for two reasons: (1) they group data together into common directions and (2) they split off major query classes. As shown in Figure 2.2, most data types have a companion web type (i.e., in, inweb, out, 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.

Chapter 3

Essential SiLK Tools

This chapter describes analyses with the six fundamental SiLK tools: `rwfilter`, `rwstats`, `rwcount`, `rwcut`, `rwsort`, and `rwuniq`. The chapter introduces these tools through example analyses, with their more general usage briefly described. At the conclusion of this chapter, you will be able to

- use `rwfilter` to choose flow records
- describe the basic partitioning parameters, including how to express IP addresses, times, and ports
- perform and display basic analyses using the SiLK tools and a shell scripting language

During this chapter, the most commonly-used parameters specific to each of these tools are covered. Section 3.9 surveys features, including parameters, that are common across several tools.

3.1 Suite Introduction

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,[8] both unnamed and named, or with intermediate files.

Flow analysis is generally input/output 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.

[8] See Section 1.2.2.

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.

3.2 Choosing Records with `rwfilter`

`rwfilter` is the most commonly used SiLK command and serves as the starting point for most analyses (as shown in the examples that follow). It both retrieves data and partitions data to isolate flow records of interest. Also it has more options (by far) than any other command in the SiLK tool suite. These options have grown as the tool has matured, driven by users' needs for more expressiveness in record filtering.

Most of the time, `rwfilter` is used in conjunction with other analysis tools or with additional invocations of `rwfilter`. However, it is also a very useful analytical tool on its own. As a simple example, consider Example 3.1, which uses `rwfilter` to print volume information on traffic from the enterprise network to an external network of interest over an eight-hour period.[9] The results show that the enterprise network sent 47,357 flows to the external network, covering an aggregate of 438,635 packets containing a total of 198,665,838 bytes. Values like these form a basis for tracking traffic over time to the external network.

Example 3.1: Using `rwfilter` to Count Traffic to an External Network

```
<1>$ rwfilter --start-date=2014/02/05T00 --end-date=2014/02/05T07 \
              --type=out,outweb --daddress=157.166.0.0/16 --print-volume-statistics
     |              Recs|           Packets|               Bytes|     Files|
Total|          32685077|        1969054465|       2199144789301|        16|
 Pass|             47357|            438635|           198665838|          |
 Fail|          32637720|        1968615830|       2198946123463|          |
```

Although parameter ordering is highly flexible, a high-level view of the `rwfilter` command is

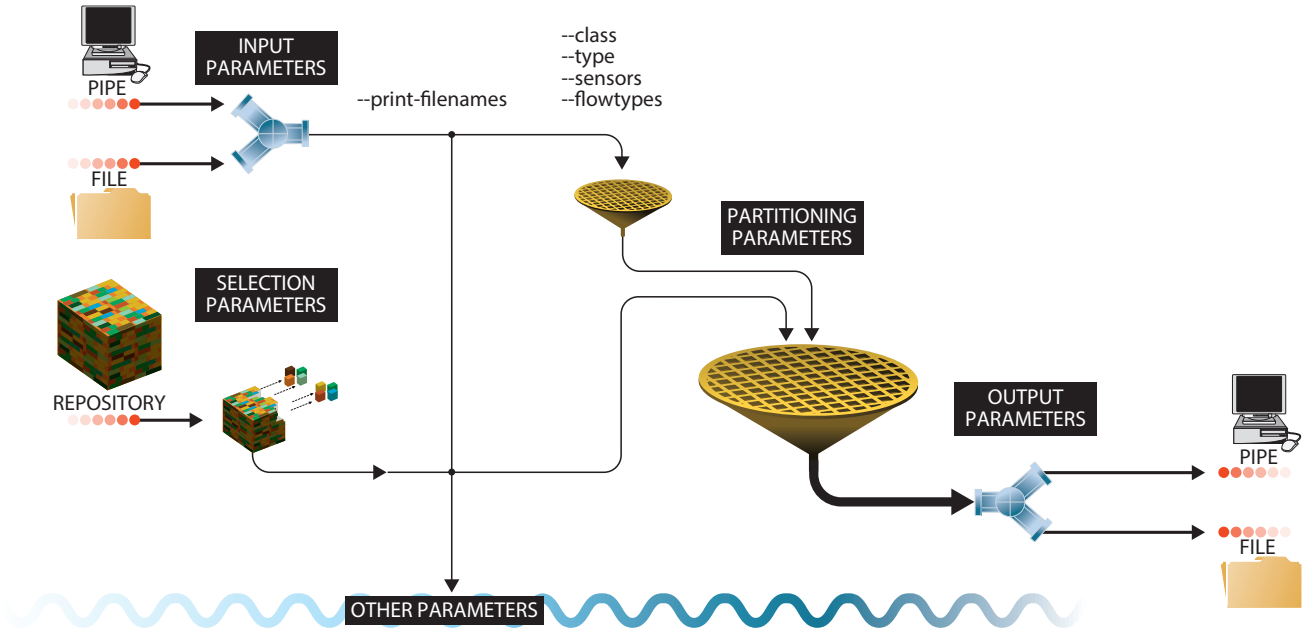
`rwfilter` {*selection* | *input*} *partition output* [*other*]

Figure 3.1 shows a high-level abstraction of the control flows in `rwfilter`, as affected by its parameters. The input to `rwfilter` is specified using either selection parameters or input parameters. The absence or presence of *input* parameters determines whether flow records will be read from the SiLK flow repository (which were created by flow sensors and stored by `rwflowpack`) or from pipes and/or named files in working directories containing records previously extracted from the repository or created by other means. In the absence of input parameters, *selection* parameters must be supplied, indicating that `rwfilter` will be reading flow records from the SiLK repository (often called a data pull). Selection parameters inform `rwfilter` which files to read from the repository, not by filename, but by supplying desired attributes of the records stored in the files. The only attributes that may be specified with selection parameters are those used to organize flow records into directories and files, as shown in Table 3.1. When specifying selection parameters, experienced analysts include `--start-date` to avoid having `rwfilter` implicitly pull all records from the current day, potentially leading to inconsistent results over time. The *other* parameter `--print-filenames` causes `rwfilter` to list, on the standard error file, the name of each file as it is opened for reading. This not only provides assurance that the expected files were read, but also provides an indication of progress (useful when many files are used as data sources and the command will take a long time to complete). *Partitioning* parameters specify

[9] Commands and results in this handbook have been anonymized to protect the privacy of the enterprise network.

which records "pass" the filter and which "fail." The main effort in composing an `rwfilter` command often lies in the specification of the partitioning parameters as `rwfilter` supports a very rich library of them. When using input parameters (i.e., when reading from pipes and/or named files), a restricted group of what are normally selection parameters may be used as partitioning parameters, as shown in the small filter grate in Figure 3.1. One, both, or neither of the sets of "pass" and "fail" records can be written to pipes and/or named files in a working directory via the *output* parameters. Lastly, there are *other* parameters (e.g., `--help`) that give useful information without accessing flow records or that modify the resources used by `rwfilter` in a given invocation.

Figure 3.1: `rwfilter` Parameter Relationships



The call to `rwfilter` in the simple example presented (Example 3.1) uses selection, partitioning, and output parameters (the minimum required by `rwfilter`). The selection parameters used in the example access all outgoing repository files for all sensors in the default class that describe flows starting between 00:00:00 and 07:59:59.999 UTC on February 5, 2014. The `--daddress` parameter is a partitioning parameter that specifies which records from the selected files "pass" and which "fail." The `--print-volume-statistics` parameter is an output parameter.

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, the processing of 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.

3.2.1 Using `rwfilter` Parameters to Control Filtering

The source of flow records may be a SiLK repository, pipes from other processes, or free-standing disk files with records previously extracted from the repository or created by other means. *Input* parameters specify the names of free-standing disk files and/or pipes (`stdin` for the unnamed pipe) from which `rwfilter` obtains the flow records. When the repository is the source of flow records, `rwfilter` invocations use selection parameters instead of input parameters. Selection parameters are mutually exclusive with input parameters. The available forms of input parameters are

- 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 SiLK Version 3 a previously required parameter named `--input-pipe` is deprecated. Use of this parameter prevents other files or pipes from being used in the command, so its use should be avoided. However, many legacy invocations of `rwfilter` may include it.

- `--xargs` parameter to specify a file containing filenames from which to read flow records. If the number of files exceeds what is convenient to put in the command line, use of `--xargs` is recommended. This parameter also is used when another UNIX process is generating the list of input files, as in

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

Selection parameters (described in Table 3.1) tell `rwfilter` from which files in the repository to pull data. In Example 3.1, the call to `rwfilter` uses three selection parameters: `--start-date`, `--end-date`, and `--type` (`--class` is left to its default value, which in many implementations is `all`; `--sensors` also is left to its default value, which is all sensors of the class). The `--start-date` and `--end-date` parameters specify that this pull applies to eight hours worth of traffic: 00:00:00 UTC to 07:59:59.999 UTC on February 5, 2014 (the parameters to `--start-date` and `--end-date` are inclusive and may be arbitrarily far apart, depending on what dates are present in the repository, although neither should be set to the future). `--start-date` and `--end-date` may specify just dates, in which case they denote whole days, or they additionally may specify an hour, in which case they denote whole hours. Note particularly that the day or hour specified in `--end-date` is not a point in time; it is a whole day or hour.

In Example 3.1, the `--type` parameter specifies that outgoing flow records are to be pulled within the specified time range. Each unique combination of selection parameters (root directory, class, sensor, type, and time) maps to one or more flow record files in the repository (depending on the number of hours included in the time). In this example, 16 files are accessed. Specifying more restrictive selection parameters results in less data being examined and thus faster queries. Be sure to understand what traffic is included in each available class and type, include all relevant types in any query, and exclude irrelevant types for improved performance. `--flowtypes` is used to specify queries across multiple classes, while restricting the types of interest on each class. Use this parameter carefully, as it is easy to specify *lots* of records to filter, which subsequently reduces performance.

Partitioning parameters define tests that divide the input records into two groups: (1) "pass" records, which satisfy all the tests and (2) "fail" records, which fail to satisfy at least one of the tests. Each call to `rwfilter` must have at least one partitioning parameter unless the only output parameter is `--all-destination`, which does not require the difference between passing and failing to be defined. In Example 3.1, flow records to a specific network are desired, so the call uses a `--daddress` parameter with an argument of CIDR block 157.166.0.0/16 (the specific network). Since even the `--print-statistics` and `--print-volume-statistics` output parameters require tests, it is a best practice to specify at least one partitioning parameter always, even when all the records read from the sources should pass the filter. When all input records should pass,

Table 3.1: `rwfilter` Selection Parameters

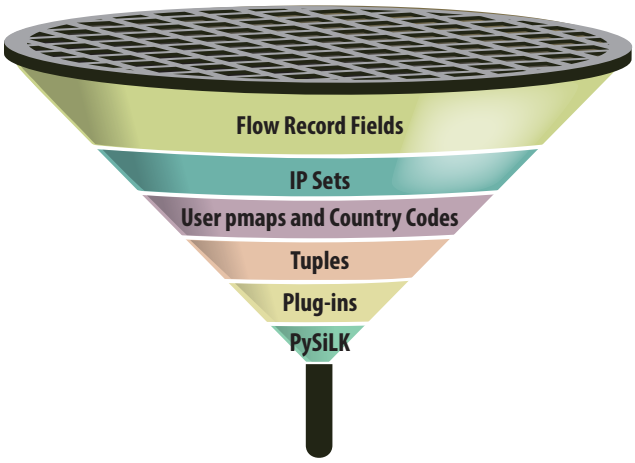
Parameter	Example	Description
`--data-rootdir`	/datavol/repos	Location of SiLK repository
`--sensors`*	1–5	Sensor(s) used to collect data
`--class`*	all	Category of sensors
`--type`*	inweb,in,outweb,out	Category of flows within class
`--flowtypes`*	c1/in,c2/all	Class/type pairs
`--start-date`	2014/6/13	First day or hour of data to examine
`--end-date`	2014/3/20T23	Final day or hour of data to examine

*These selection keywords may be used as partitioning options if an input file or pipe is named.

by convention an analyst uses `--protocol` with an argument of `0-` (meaning zero or higher), which is a test that can be met by all IP traffic, since no protocol numbers are lower than zero.[10]

Partitioning parameters are the most numerous in order to provide great flexibility in describing which flow records pass or fail. Later examples will show several partitioning parameters. (See `rwfilter --help` for a full listing; many of the more commonly used parameters are listed in Tables 3.2–3.8.) As shown in Figure 3.2, there are several groups of partitioning parameters. This section focuses on the parameters that partition records based on fields in the flow records. Section 4.4 discusses IP sets and how to filter with those sets. Section 4.7 describes prefix maps and country codes. Section 3.2.6 discusses tuple files and the parameters that use them. The use of plug-ins is dealt with in Section 4.8. Lastly, Section 5.2 describes the use of PySiLK plug-ins. Figure 3.2 also illustrates the relative efficiency of the types of partitioning parameters. Choices higher in the illustration may be more efficient than choices lower down. So analysts should prefer IP sets to tuple files when either would work, and they should prefer `--saddress` or `--scidr` to IP sets when those would all work.

Figure 3.2: `rwfilter` Partitioning Parameters



[10]See http://www.iana.org/assignments/protocol-numbers; in IPv4 this is the *protocol* field in the header, but in IPv6 this is the *next-header* field—both have the range 0–255.

Table 3.2: Single-Integer- or Range-Partitioning Parameters

Parameter	Example	Partition Based on
`--packets`	1–3	Packet count in flow
`--bytes`	400–2400	Byte count in flow
`--bytes-per-packet`	1000–1400	Average bytes/packet in flow
`--duration`	1200–	Duration of flow in seconds
`--ip-version`	6	IP version of the flow record

Table 3.3: Multiple-Integer- or Range-Partitioning Parameters

Parameter	Example	Partition Based on
`--protocol`	0,2–5,7–16,18–	Protocol number (6=TCP, 17=UDP, 1=ICMP)
`--sport`	0–1023	Source port
`--dport`	25	Destination port
`--aport`	80,8080	Any port. Like `--sport`, but for either source or destination
`--application`	2427,2944	Application-layer protocol

Table 3.4: Address-Partitioning Parameters

Parameter	Example	Partition Based on
`--saddress`	198.51.100.1,254	Single address, CIDR block, or wildcard for source
`--daddress`	198.51.100.0/24	Like `--saddress`, but for destination
`--any-address`	2001:DB8::x	Like `--saddress`, but for either source or destination
`--next-hop-id`	10.2–5.x.x	Like `--saddress`, but for next hop address
`--scidr`	198.51.100.1,203.0.113.64/29	Multiple addresses and CIDR blocks for source address
`--dcidr`	FC00::/7,2001:DB8::/32	Like `--scidr`, but for destination
`--any-cidr`	203.0.113.199,192.0.2.44	Like `--scidr`, but for either source or destination
`--nhcidr`	203.0.113.8/30,192.0.2.6	Like `--scidr`, but for next hop address
`--sipset`	tornodes.set	Source address existing in IP set file
`--dipset`	websvrs.set	Destination address existing in IP set file
`--anyset`	dnssvrs.set	Either source or destination address in IP set file
`--nhipset`	lgvolume.set	Next hop address in IP set file
`--not-`*param*	Any address parameter can be inverted using the `--not-` prefix	

Table 3.5: High/Mask Partitioning Parameters

Parameter	Example	Partition Based on
`--flags-all`	SF/SF,SR/SR	Accumulated TCP flags
`--flags-initial`	S/SA	TCP flags in the first packet of flow
`--flags-session`	/FR	Flags in the packets after the first
`--attributes`	T/T,C/C	Termination attributes or packet size uniformity

Table 3.6: Time-Partitioning Parameters

Parameter	Example	Partition Based on
`--stime`	2014/4/7–2014/4/8T12	Flow's start time
`--etime`	2014/4/7T8:30–2014/4/8T8:59	Flow's end time
`--active-time`	2014/4/7T11:33:30–	Overlap of active range and period between flow's start and end times

Table 3.7: Country-Code-Partitioning Parameters

Parameter	Example	Partition Based on
`--scc`	ru,cn,br,ko	Source address's country code
`--dcc`	ca,us,mx	Destination address's country code
`--any-cc`	a1,a2,o1,--	Source or destination address's country code

Table 3.8: Miscellaneous Partitioning Parameters

Parameter	Example	Partition Based on
`--pmap-src-`*mapname*	Zer0n3t,YOLOBotnet	Source mapping to a specified label
`--pmap-dst-`*mapname*	DMZ,bizpartner	Destination mapping to a specified label
`--pmap-any-`*mapname*	mynetdvcs	Source or destination mapping to a specified label (see Section 4.7 on page 127)
`--tuple-file`	torguard.tuple	Match to any specified combination of field values (see Section 3.2.6 on page 42)
`--python-expr`	`'rec.sport==rec.dport'`	Truth of expression (see Section 5.2 on page 138)
`--python-file`	complexrules.py	Truth of value returned by the program in the Python file (see Section 5.2 on page 138)
`--plugin`	flowrate.so	Custom partitioning with a C language program (see Section 4.8 on page 133)

Partitioning parameters specify a collection of flow record criteria, such as the protocols 6 and 17 or the specific IP address 198.51.100.71. As a result, almost all partitioning parameters describe some group of values. These ranges are generally expressed in the following ways:

Value range: Value ranges are used when all values in a closed interval are desired. A value range is two numbers separated by a hyphen, such as `--packets=1-4`, which indicates that flow records with a packet count from one through four (inclusive) belong to the *pass* set of records. Some partitioning parameters (such as `--duration`) only make sense with a value range (searching for flows with a duration exact to the millisecond would be fruitless); An omitted value on the end of the range (e.g., `--bytes=2048-`) specifies that any value greater than or equal to the low value of the range passes. Omitting the value at the start of a range is not permitted. The options in Table 3.2 should be specified with a single value range, except `--ip-version` which accepts a single integer.

Value alternatives: Fields that have a finite set of values (such as ports or protocol) can be expressed using a comma-separated list. In this format a field is expressed as a set of numbers separated by commas. When only one value is acceptable, it is presented without a comma. Examples include `--protocol=3` and `--protocol=3,9,12`. Value ranges can be used as elements of value alternative lists. For example, `--protocol=0,2-5,7-16,18-` says that all flow records that are not for ICMP (1), TCP (6), or UDP (17) traffic are desired. The options in Table 3.3 are specified with value alternatives.

IP addresses: IP address specifications are expressed in three ways: a single address, a CIDR block (a network address, a slash, and a prefix length), and a SiLK address wildcard. The `--saddress`, `--daddress`, `--any-address`, and `--next-hop-id` options, and the `--not-` forms of those options accept a single specification that may be any of three forms. The `--scidr`, `--dcidr`, `--any-cidr`, and `--nhcidr` options, and the `--not-` forms of those options accept a comma-separated list of specifications that may be addresses or CIDR blocks but not wildcards. SiLK address wildcards are address specifications with a syntax unique to the SiLK tool suite. Like CIDR blocks, a wildcard specifies multiple IP addresses. But while CIDR blocks only specify a continuous range of IP addresses, a wildcard can even specify discontiguous addresses. For example, the wildcard `1-13.1.1.1,254` would select the addresses 1.1.1.1, 2.1.1.1, and so on until 13.1.1.1, as well as 1.1.1.254, 2.1.1.254, and so on until 13.1.1.254. For convenience, the letter `x` can be used to indicate all values in a section (equivalent to `0-255` in IPv4 addresses, `0-FFFF` in IPv6 addresses). CIDR notation may also be used, so `1.1.0.0/16` is equivalent to `1.1.x.x` and `1.1.0-255.0-255`. Any address range that can be expressed with CIDR notation also can be expressed with a wildcard. Since CIDR notation is more widely understood, it probably should be preferred for those cases. As explained in Section 1.1.3, IPv6 addresses use a double-colon syntax as a shorthand for any sequence of zero values in the address and use a CIDR prefix length. The options in Table 3.4, except the set-related options, are specified with IP addresses. The set-related options specify a filename.

TCP flags: The `--flags-all`, `--flags-initial`, and `--flags-session` options to `rwfilter` use a compact, yet powerful, way of specifying filter predicates based on the presence of TCP flags. The argument to this parameter has two sets of TCP flags separated by a forward slash (`/`). To the left of the slash is the *high* set; it lists the flags that must be set for the flow record to pass the filter. The flag set to the right of the slash contains the *mask*; this set lists the flags whose status is of interest, and the set must be non-empty. Flags listed in the mask set but not in the high set must be off to pass. The flags listed in the high set must be present in the mask set. (For example, `--flags-initial=S/SA` specifies a filter for flow records that initiate a TCP session; the S flag is high [on] and the A flag is low [off].) See Example 3.2 for another sample use of this parameter. The options in Table 3.5, except `--attributes`, are specified with TCP flags; `--attributes` also specifies flags in a high/mask format, but the flags aren't TCP flags.

Attributes: The `--attributes` parameter takes any combination of the letters `S`, `T`, and `C`, expressed in high/mask notation just as for TCP flags. `S` indicates that all packets in the flow have the same length (never present for single-packet flows). `T` indicates the collector terminated the flow collection due to active timeout. `C` indicates the collector produced the flow record to continue flow collection that was terminated due to active timeout. Only the `--attributes` parameter uses attributes for values.

Time ranges: Time ranges are two times, potentially precise to the millisecond, separated by a hyphen; in SiLK, these times can be expressed in their full *YYYY/MM/DD*`T`*hh:mm:ss.mmm* form (e.g., `2005/02/11T03:18:00.005-2005/02/11T05:00:00.243`). The times in a range may be abbreviated by omitting a time (but not date) component and all the succeeding components. If all the time components are omitted, the `T` (or colon) that separates the time from the date may also be omitted. Abbreviated times (times without all the components down to the millisecond) are treated as though the last component supplied includes the entire period specified by that component, not just an instant (i.e., if the last component is the day, it represents a whole day; if it's the hour, it represents the whole hour.) So `2014/1/31` represents one whole day. `2014/1/31-2014/2/1` represents two days. `2014/1/31T14:50-2014/1/31T14:51` represents two whole minutes. `2014/1/31T12-` represents all time from 2014/1/31 noon forward. The options in Table 3.6 are specified with time ranges.

Country codes: The `--scc`, `--dcc`, and `--any-cc` parameters take a comma-separated list of two-letter country codes, as specified by IANA.[11] There are also four special codes: `--` for unknown, `a1` for anonymous proxy, `a2` for satellite provider, and `o1` for other. The options in Table 3.7 are specified with country codes.

Output parameters to `rwfilter` specify which statistics or sets of records should be returned from the call. There are five output parameters, as described in Table 3.9. Each call to `rwfilter` must have at least one of these parameters. In Example 3.1, the `--print-volume-statistics` option is used to count the flow records and their associated byte and packet volumes. When `--print-statistics` or `--print-volume-statistics` is used without a value, the equal sign between the parameter name and the value also is omitted.

Table 3.9: `rwfilter` Output Parameters

Parameter	Example	Description
`--pass-destination`	stdout	Send SiLK flow records matching all partitioning parameters to pipe or file
`--fail-destination`	faildata.rw	Like `--pass`, but for records failing to match
`--all-destination`	allrecs.rw	Like `--pass`, but for all records
`--print-statistics`	—	Print (default to stderr) count of records passing and failing
`--print-volume-statistics`	out-vol.txt	Print counts of flows/bytes/packets read, passing, and failing to named file or stderr

One of the most useful tools available for in-depth analysis is the drilling-down capability provided by using the `rwfilter` parameters `--pass-destination` and `--fail-destination`. These two parameters are so commonly used that the remainder of this handbook will refer to them by their common abbreviations, `--pass` and `--fail`. Most analysis will involve identifying an area of interest (all the IP addresses that communicate with address x, for example) and then combing through those data. Rather than pulling down the same base query repeatedly, store the data to a separate data file using the `--pass` switch.

[11] http://www.iana.org/domains/root/db/

Occasionally, it is more convenient to describe the data *not* wanted than the desired data. The `--fail` option allows saving data that doesn't match the specified conditions.

An interesting construct, called a *manifold*, uses several `rwfilter` commands to categorize traffic (see Figure 3.3). If the categories are non-overlapping, the manifold uses the `--pass` parameter to store the matching traffic in a category and uses the `--fail` parameter to transfer the other traffic to the next `rwfilter` command for the next category. The last `rwfilter` command saves the last category and uses the `--fail` parameter to save the remaining uncategorized traffic.

Figure 3.3: A Manifold

A manifold for *overlapping* categories again would use the `--pass` parameter for traffic in a category, but would use `--all-destination` (`--all` for short) to transfer *all* traffic to the next `rwfilter`. All the traffic is transferred to the next `rwfilter` because with overlapping categories, the manifold is not done with a record just because it assigned a first category to the record. The record may belong to other categories as well.

Other parameters are miscellaneous parameters to `rwfilter` that have been found to be useful in analysis or in maintaining the repository. These are somewhat dependent on the implementation and include those described in Table 3.10. None of these parameters are used in the example, but at times they are quite useful.

The `--threads` parameter takes an integer scalar N to specify using N threads to read input files and filter records. The default value is one or the value of the `SILK_RWFILTER_THREADS` environment variable if that is set. Using multiple threads is preferable for queries that look at many files but return few records. Current experience is that performance peaks at about two to three threads per CPU (core) on the host running `rwfilter`, but this result is variable with the type of query and the number of records returned from each file. There is no advantage in having more threads than input files. There may also be a point of diminishing

Table 3.10: Other Parameters

Parameter	Description
`--print-missing-files`	Print names of missing repository files to stderr. Doubles as a selection parameter.
`--threads`	Specify number of process threads to be used in filtering
`--max-pass-records`	Specifies the maximum number of records to return as matching partitioning parameters
`--max-fail-records`	Specifies the maximum number of records to return as *not* matching partitioning parameters
For additional parameters, see Table 3.13 on page 71 and Table 3.14 .	

returns, which seems to be around 20 threads.

To improve query efficiency when few records are needed, the `--max-pass-records` parameter allows the analyst to specify the maximum number of records to return via the path specified by the `--pass` parameter.

3.2.2 Finding Low-Packet Flows with `rwfilter`

The TCP state machine is complex (see Figure 1.4 on page 12). As described in Section 1.1.4, legitimate service requests require a minimum of three (more commonly four) packets in the flow from client to server. There may be only two packets in the flow from server to client. There are several types of illegitimate traffic (such as port scans and responses to spoofed-address packets) that involve TCP flow records with low numbers of packets. Occasionally, there are legitimate TCP flow records with low numbers of packets (such as continuations of previously timed-out flow records, contact attempts on hosts that don't exist or are down, services that are not configured, and RST packets on already closed connections), but this legitimate behavior is relatively rare. As such, it may be useful to understand where low-packet TCP network traffic comes from and when such flow records are collected most frequently.

Example 3.2 shows more complex calls to `rwfilter`. The call to `rwfilter` in this example's Command 1 selects all incoming flow records in the repository that started from 00:00:00 UTC to 05:59:59.999 UTC, that describe TCP traffic, and that had three or fewer packets in the flow record. The call in Command 2 passes flow records that had the SYN flag set in any of their packets, but the ACK, RST, and FIN flags not set in any of their packets, and fails those that did not show this flag combination. The third call extracts the flow records that have the RST flag set, but had the SYN and FIN flags not set.

The calls in Commands 2 and 3 of Example 3.2 use a filename as an input parameter; in each case, a file produced by a preceding call to `rwfilter` is used. These commands show how `rwfilter` can be used to refine selections to isolate flow records of interest. The call in Command 1 is the only one that pulls from the 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. The calls in Commands 2 and 3 use `--flags-all` as a partitioning parameter to pull out flow records of interest. All three calls use `--pass` output parameters, and the call in Command 2 also uses a `--fail` output parameter to generate a temporary file that serves as input to Command 3.

Example 3.2: Using `rwfilter` to Extract Low-Packet Flow Records

```
<1>$ rwfilter --start-date=2014/02/05T00 --end-date=2014/02/05T05 \
              --type=in,inweb --protocol=6 --packets=1-3 \
              --print-statistics --pass=lowpacket.rw
Files     12.  Read    71376178.  Pass    28945230. Fail     42430948.
<2>$ rwfilter lowpacket.rw --flags-all=S/SARF \
              --print-statistics --pass=synonly.rw --fail=temp.rw
Files      1.  Read    28945230.  Pass    23436799. Fail      5508431.
<3>$ rwfilter temp.rw --flags-all=R/SRF \
              --print-statistics --pass=reset.rw
Files      1.  Read     5508431.  Pass     2065861. Fail      3442570.
```

3.2.3 Using IPv6 with `rwfilter`

IPv6 is fully supported by SiLK version 3, as long as SiLK was installed with IPv6 support. Example 3.3 shows the `--ip-version` option being used to partition records based on the IP version of each record.

Example 3.3 shows port numbers. The ports for the UDP (protocol 17) flow are sensible: specifically, 5353 is Multicast DNS. But the ports for protocol 58 look odd. Protocol 58 is ICMPv6. As with ICMPv4 (protocol 1), ICMPv6 uses types and codes to categorize ICMP messages. Types and codes are one byte each. Since ICMP doesn't use ports, SiLK saves space in the flow record by repurposing the two-byte destination port field to hold the ICMP type and code. Looking at the dPort field as a decimal value isn't too revealing as to the encoded ICMP type and code. The next example shows how to display that information more meaningfully.

Example 3.3: Using `rwfilter` to Partition Flows on IP Version

```
<1>$ rwfilter --start-date=2012/4/1T0 --type=in --ip-version=6 \
              --print-statistics --pass=stdout \
      | rwcut  --fields=1-5
Files      1.  Read        1457.  Pass           5. Fail         1452.
                         sIP|                          dIP|sPort|dPort|pro|
  fe80::217:f2ff:fed4:308c|                      ff02::fb| 5353| 5353| 17|
  fe80::213:72ff:fe95:31d3|                       ff02::1|    0|34304| 58|
  fe80::213:72ff:fe95:31d3|          ff02::1:ffce:93a5|    0|34560| 58|
  fe80::213:72ff:fe95:31d3|2001:db8:1::a0ff:fece:93a5|    0|34560| 58|
  fe80::213:72ff:fe95:31d3|                       ff02::1|    0|34304| 58|
```

Example 3.4 shows how to detect neighbor discovery solicitations and advertisements in IPv6 data. Such solicitations (ICMPv6 type 135) request, among other things, the network interface address for the host with the given IPv6 address—serving the function of IPv4's address resolution protocol. Network discovery advertisements (ICMPv6 type 136) are the responses with this information. See the last two lines of output in Example 3.4 for an example solicitation with its responding advertisement. The example shows the use of the field names `iType` and `iCode`, which were introduced with SiLK Version 3.8.1. With an older SiLK version, use the single name `icmpTypeCode` to produce the same two fields.

Since Example 3.4 doesn't partition on IP version, there is the possibility of retrieving IPv4 records as well as, or instead of, IPv6 records. Also, IPv6 records can contain IPv4 addresses in the form of IPv4-compatible

Example 3.4: Using `rwfilter` to Detect IPv6 Neighbor Discovery Flows

```
<1>$ rwfilter --start-date=2013/4/1T1 --icmp-type=135,136 --pass=stdout \
      | rwcut --fields=1,2,5,iType,iCode
                         sIP|                         dIP|pro|iTy|iCo|
  fe80::213:72ff:fe95:31d3|          ff02::1:ffd4:308c| 58|135|  0|
  fe80::213:72ff:fe95:31d3|2001:db8:1::a0ff:fece:93a5| 58|135|  0|
  fe80::213:72ff:fe95:31d3|          ff02::1:ffce:93a5| 58|135|  0|
2001:db8:1::a0ff:fece:93a5|  fe80::213:72ff:fe95:31d3| 58|136|  0|
```

or IPv4-mapped addresses. Section 3.9.4 describes how to control the filtering and display of IP addresses of both versions.

3.2.4 Using Pipes with `rwfilter` to Divide Traffic

One problem with generating temporary files is that they are slow. A first command writes all the data to disk, and then a subsequent command reads the data back from disk. A faster method is using UNIX pipes to pass records from one process to another, which allows tools to operate concurrently, using memory (when possible) to pass data between tools. Setting up an unnamed pipe between processes is described in Section 1.2.2 on page 15.

Example 3.5 shows a call to `rwfilter` that uses an output parameter to write records to standard output, which is piped to a second call to `rwfilter` that reads these records via standard input. The first call pulls repository records describing incoming traffic that transferred 2,048 bytes or more. The second call (after the pipe) partitions these records for traffic that takes 20 minutes (1,200 seconds) or more and for traffic that takes less than 20 minutes. (Recall that 20 minutes is close to the maximum duration of flows in many configurations; traffic much longer than 20 minutes will be split by the collection system.)

Example 3.5: `rwfilter --pass` and `--fail` to Partition Fast and Slow High-Volume Flows

```
<1>$ rwfilter --start-date=2014/02/05T00 --end-date=2014/02/05T05 \
              --type=in,inweb --bytes=2048- --pass=stdout \
      | rwfilter stdin --duration=1200- \
                  --pass=slowfile.rw --fail=fastfile.rw
<2>$ ls -l slowfile.rw fastfile.rw
-rw------- 1 user1 group1 83886269 May  5 20:05 fastfile.rw
-rw------- 1 user1 group1  4456015 May  5 20:05 slowfile.rw
```

3.2.5 Translating IDS Signatures into `rwfilter` Calls

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, but it is possible to identify specific intrusion tools using SiLK. Intruders generally use automated tools or worms to infiltrate networks. While directed intrusions are still a threat, tool-based, broad-scale intrusions are more common. Sometimes it will be necessary to translate a signature into filtering rules, and this section describes some standard guidelines.

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. If a specific size can be identified, that can be used to identify tools. When working with packet sizes, remember that the SiLK suite includes the packet headers, so a 376 byte UDP payload, for example, 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, 14 packets with a total size of 2,080 bytes).

In addition, the `rwidsquery` tool takes as input either a SNORT® alert log or rule file, analyzes the contents, and invokes `rwfilter` with the appropriate arguments to retrieve flow records that match attributes of the input file.

3.2.6 Using Tuple Files with `rwfilter` for Complex Filtering

For a variety of analyses, the partitioning criteria are specific combinations of field values, any one of which should be considered as passing. While the analyst can do this via separate `rwfilter` calls and merge them later, this can be inefficient as it may involve pulling the same records from the repository several times.

In particular, what won't work is anything like this

```
rwfilter --protocol=6,17 --dport=25,161
```

which makes four permutations and not just the two that apply in this example: SMTP (protocol 6 [TCP], port 25) and SNMP (protocol 17 [UDP], port 161).

The analyst could pull each choice separately, but that would reread the input flow records. If the analysis demands many cases, the same data would be read many times:

```
rwfilter --protocol=6 --dport=25 --pass=result.rw
rwfilter --protocol=17 --dport=161 --pass=part2.rw
rwappend result.rw part2.rw
rm part2.rw
```

A more efficient solution is to store the partitioning criteria as a *tuple file* and use that file with `rwfilter` to pull all the records in a single operation.

Example 3.6 shows a tuple file used to choose some addresses with one port and other addresses with another port. Command 1 creates the tuple file, which is a text file consisting of flow fields delimited by vertical bars. The first line contains headers identifying the fields associated with the columns. Then this file can 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.

Example 3.6: `rwfilter` with a Tuple File

```
<1>$ cat <<END_FILE >ssh-smtp.tuple
            dIP|dPort
 192.168.132.72|   22
192.168.145.167|   22
192.168.207.184|   22
192.168.135.207|   22
192.168.206.210|   22
 192.168.160.50|   25
 192.168.46.248|   25
 192.168.97.150|   25
192.168.100.252|   25
 192.168.129.25|   25
END_FILE
<2>$ rwfilter --start-date=2014/02/05T00 --end-date=2014/02/05T03 \
              --type=in --protocol=6 --tuple-file=ssh-smtp.tuple --pass=stdout \
      | rwuniq --fields=dIP,dPort --value=Records
            dIP|dPort|   Records|
 192.168.160.50|   25|     95997|
 192.168.129.25|   25|      8072|
 192.168.132.72|   22|      2544|
192.168.206.210|   22|      2267|
192.168.135.207|   22|      2328|
192.168.100.252|   25|     20415|
192.168.207.184|   22|      2538|
 192.168.97.150|   25|     30285|
 192.168.46.248|   25|     30922|
192.168.145.167|   22|      2540|
```

It also is possible to obtain quicker results by using seemingly redundant command-line options to duplicate some values from the tuple file. Adding `--daddress=192.168.0.0/16` and/or `--dport=22,25` may reduce the number of records that need to be examined with the tuple file. Note, however, that these options are not a substitute for the tuple file in this example, as they are not sufficiently restrictive to produce the desired results.

3.3 Describing Flows with `rwstats`

`rwstats` provides a collection of statistical summary and counting facilities that enable the organizing and ranking of traffic according to various attributes. `rwstats` operates in one of two modes: top/bottom-*N* or protocol distribution statistics.

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`, `--percentage`, or `--threshold`
- one of the summary parameters (`--overall-stats` or `--detail-proto-stats`)

3.3.1 Examining Extremes with `rwstats` Top or Bottom-*N* Mode

These statistics can be collected for a single flow field or any combination of flow fields. Example 3.7 illustrates three calls to `rwstats`. Command 1 generates a count of flow records for the top five protocols. In this case, the fifth protocol has a cumulative percentage of 100 displayed, indicating that there are no additional protocols after this one (also, the line in the listing labeled `INPUT` shows that only five bins were produced). Command 2 uses `rwfilter` to extract all UDP flow records from the file and pass them along to a second call to `rwstats`, which displays the top five destination ports. Command 3 does a combination of protocol and dPort with `rwstats` to generate the five most common protocol/port pairs, which include protocol 1, ICMPv4 (that appears with a non-zero port number, although ICMP does not use ports). This is because flow generators encode the ICMP message type and code in the dPort field of NetFlow and IPFIX records.

`rwstats` in top/bottom-*N* mode produces a columnar table. The first fields are key fields, followed by aggregate counts such as records, packets, or bytes and then the percentage contribution of the key value to the total for the primary (first) aggregate count. The final column is a cumulative percentage—the percentage of all values of the total set—up to this key value. As with other SiLK applications, `rwstats` can read its input either from a file or a pipe, as shown in Example 3.7.

Key fields are chosen with the `--fields` option. Aggregate count fields are specified with the `--values` option. The default for `--values` is `Records` (alias `Flows`). Other choices are `Packets`, `Bytes`, and `Distinct:`*key-field*. The last option is a count of distinct (unique) values for the *key-field*, which cannot also be present in the `--fields` list. There are legacy synonyms for `Distinct:sIP` and `Distinct:dIP`, which are `sIP-Distinct` and `dIP-Distinct`, respectively.

`rwstats` can display either the keys with the *N* greatest primary aggregate values with the `--top` (default) parameter or the keys with the *N* lowest primary aggregate values with the `--bottom` parameter. Figure 3.4 provides a brief summary of `rwstats` and its more common parameters.

Example 3.7: Using `rwstats` to Count Protocols and Ports

```
<1>$ rwstats slowfile.rw --fields=protocol --values=Records --count=5
INPUT: 160919 Records for 5 Bins and 160919 Total Records
OUTPUT: Top 5 Bins by Records
pro|   Records|  %Records|   cumul_%|
  6|    104917| 65.198640| 65.198640|
 17|     48235| 29.974708| 95.173348|
  1|      7220|  4.486729| 99.660077|
 50|       490|  0.304501| 99.964578|
 47|        57|  0.035422|100.000000|
<2>$ rwfilter slowfile.rw --protocol=17 --pass=stdout \
      | rwstats --fields=dPort --values=Records --count=5
INPUT: 48235 Records for 269 Bins and 48235 Total Records
OUTPUT: Top 5 Bins by Records
dPort|   Records|  %Records|   cumul_%|
  123|     38020| 78.822432| 78.822432|
  137|      4386|  9.092982| 87.915414|
   53|      2075|  4.301855| 92.217270|
 4500|      1218|  2.525137| 94.742407|
  443|       342|  0.709029| 95.451436|
<3>$ rwstats slowfile.rw --fields=protocol,dPort --values=Records --count=5
INPUT: 160919 Records for 7130 Bins and 160919 Total Records
OUTPUT: Top 5 Bins by Records
pro|dPort|   Records|  %Records|   cumul_%|
  6|  443|     64107| 39.838055| 39.838055|
 17|  123|     38020| 23.626794| 63.464849|
  1| 2048|      6286|  3.906313| 67.371162|
 17|  137|      4386|  2.725595| 70.096757|
 17|   53|      2075|  1.289469| 71.386225|
```

Figure 3.4: Summary of `rwstats`

rwstats

Description	Summarizes SiLK flow records by one of a limited number of key-and-value pairs and displays the results as a top-*N* or bottom-*N* list
Call	`rwstats flowrecs.rw --fields=protocol --values=Records --top --count=20`
Parameters	Choose one or none: **`--top`** Prints the top *N* keys and their values (default) **`--bottom`** Prints the bottom *N* keys and their values **`--overall-stats`** Prints minima, maxima, quartiles, and interval-count statistics for bytes, packets, and bytes per packet across all flows **`--detail-proto-stats`** Prints overall statistics for each specified protocol. Protocols are specified as integers or ranges separated by commas. Choose one for `--top` or `--bottom`: **`--count`** Displays the specified number of key-and-value pairs **`--percentage`** Displays key-and-value pairs where the aggregate value is greater than (`--top`) or less than (`--bottom`) this percentage of the total aggregate value **`--threshold`** Displays key-and-value pairs where the aggregate value is greater than (`--top`) or less than (`--bottom`) the specified constant Options for `--top` or `--bottom`: **`--fields`** Uses the indicated fields as the key (see Table 3.11) – required **`--values`** Calculates aggregate values for the specified fields (default: Records) **`--presorted-input`** Assumes input is already sorted by key **`--no-percents`** Does not display percent-of-total or cumulative-percentage **`--bin-time`** Adjusts **`sTime`** and **`eTime`** to multiple of the argument in seconds **`--temp-directory`** Specifies the location for temporary data when memory is exceeded For additional parameters, see Table 3.13 on page 71 and Table 3.14.

Example 3.8 illustrates how to show all values that exceed 1% of all records using `rwstats`. In this particular case, only two key values (source port values) were used to send bulk data quickly: HTTP (80) and HTTPS (443).

Example 3.8: `rwstats --percentage` to Profile Source Ports

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwstats --fields=sPort --values=Records --top --percentage=1
INPUT: 2903475 Records for 64011 Bins and 2903475 Total Records
OUTPUT: Top 2 bins by Records (1% == 29034)
sPort|   Records|  %Records|   cumul_%|
   80|    618378| 21.297859| 21.297859|
  443|    605465| 20.853116| 42.150974|
```

As Example 3.8 indicates, distributions can be very heavily skewed. Counting the top source-port percentages in incoming traffic will skew the result towards external servers, since servers will be responding to traffic on a limited number of ports. This also is shown in the destination port count (see Example 3.9) where it is dominated by internal web servers with a smattering of email and file transfer server traffic.

Example 3.9: `rwstats --count` to Examine Destination Ports

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwstats --fields=dPort --values=Records --top --count=5
INPUT: 2903475 Records for 64493 Bins and 2903475 Total Records
OUTPUT: Top 5 Bins by Records
dPort|   Records|  %Records|   cumul_%|
  443|    728165| 25.079086| 25.079086|
   80|    690807| 23.792421| 48.871507|
   25|     18812|  0.647913| 49.519421|
   21|     18123|  0.624183| 50.143604|
   20|      9473|  0.326264| 50.469868|
```

As Example 3.9 shows, the most active destination port (HTTPS) comprises about 25% of records and, combined with nearly 24% for the next port (HTTP), results in nearly 49% of flows that are destined for internal web servers. The line above the titles starting with `INPUT:` provides a summary of the number of unique key values (bins) observed, and as this line indicates, nearly all possible destination ports (out of 65,536) are listed in this file.

For efficiency and flexibility, analysts may want to chain together `rwstats` calls or chain `rwstats` with other SiLK tools. Two parameters are used to support this in Example 3.10. `--copy-input` specifies a pipe or file to receive a copy of the flow records supplied as input to `rwstats`. `--output-path` specifies a filename to receive the textual output from the current call to `rwstats`. These parameters are also available in other tools in the suite, as shown in Table 3.13 on page 71.

Example 3.10: `rwstats --copy-input` and `--output-path` to Chain Calls

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwstats --fields=dPort --values=Records --top --count=5 \
                --copy-input=stdout --output-path=top.txt \
      | rwstats --fields=sIP --values=Records --top --count=5
INPUT: 2903475 Records for 140801 Bins and 2903475 Total Records
OUTPUT: Top 5 Bins by Records
            sIP|   Records|  %Records|   cumul_%|
 172.31.131.219|    212666|  7.324534|  7.324534|
 172.31.185.130|     56258|  1.937609|  9.262143|
  172.31.57.138|     40066|  1.379933| 10.642075|
  172.31.199.62|     39045|  1.344768| 11.986843|
   172.31.3.216|     24827|  0.855079| 12.841922|
<2>$ cat top.txt
INPUT: 2903475 Records for 64493 Bins and 2903475 Total Records
OUTPUT: Top 5 Bins by Records
dPort|   Records|  %Records|   cumul_%|
  443|    728165| 25.079086| 25.079086|
   80|    690807| 23.792421| 48.871507|
   25|     18812|  0.647913| 49.519421|
   21|     18123|  0.624183| 50.143604|
   20|      9473|  0.326264| 50.469868|
```

3.4 Creating Time Series with `rwcount`

`rwcount` provides time-binned counts of the number of records, bytes, and packets for flows occurring during each bin's assigned time period. The `rwcount` call in Example 3.11 counts into 10-minute bins the flow volume information gleaned from the `slowfile.rw` file.

By default, `rwcount` produces the table format shown in Example 3.11: the first column is the timestamp for the earliest moment in the bin, followed by the number of flow records, bytes, and packets for the bin. The `--bin-size` parameter specifies the size of the counting bins in seconds; `rwcount` uses 30-second bins by default.

SiLK doesn't have information in the flow records about the time distribution of packets and bytes in the flow. The analyst may impose a particular distribution on `rwcount` with the `--load-scheme` option in order to emphasize certain characteristics of the flows. The `--load-scheme=1` parameter specifies that all the packets and bytes in a flow should be assumed to have occurred in the first millisecond of the flow's duration, which puts all those packets and bytes into one bin, even if the flow's time period spans multiple bins. The default behavior is to distribute the packets and bytes equally among all the milliseconds in the flow's duration.

In Example 3.11, the bin (at T01:10:00), which has zero flows, packets, and bytes, has been suppressed by the `--skip-zeroes` option to reduce the length of the listing. Rows with zero flows should not be suppressed if the output is being passed to a plotting program; if they are, those data points won't be plotted.

Figure 3.5 provides a summary of this command and its options.

Example 3.11: `rwcount` for Counting with Respect to Time Bins

```
<1>$ rwcount slowfile.rw --bin-size=600 --load-scheme=1 --skip-zeroes
                Date|         Records|               Bytes|           Packets|
2014/02/05T00:00:00|         4959.00|        5367160074.00|         8243559.00|
2014/02/05T00:10:00|         5074.00|        6910309987.00|         9214692.00|
2014/02/05T00:20:00|         5062.00|        6543758658.00|        10563702.00|
2014/02/05T00:30:00|         5215.00|        3827319313.00|         8096120.00|
2014/02/05T00:40:00|         5131.00|         5781199217.00|         8176001.00|
2014/02/05T00:50:00|         1225.00|        2432680219.00|         2622752.00|
2014/02/05T01:00:00|            9.00|           1433197.00|           25572.00|
2014/02/05T01:20:00|        13554.00|       12675094915.00|        21562396.00|
2014/02/05T01:30:00|         1244.00|        1286621227.00|         2716149.00|
2014/02/05T01:40:00|         1048.00|         858182492.00|         1309776.00|
2014/02/05T01:50:00|        12432.00|       11799437751.00|        22582162.00|
2014/02/05T02:00:00|         2051.00|        1655607591.00|         3591827.00|
2014/02/05T02:10:00|         1459.00|        2038879018.00|         3961393.00|
2014/02/05T02:20:00|        10791.00|       10533713512.00|        22050491.00|
2014/02/05T02:30:00|         3117.00|        4615953786.00|         5097020.00|
2014/02/05T02:40:00|         1674.00|        1510774949.00|         3176011.00|
2014/02/05T02:50:00|         8968.00|       11837749052.00|        21935889.00|
2014/02/05T03:00:00|         4392.00|        3352830914.00|         4085321.00|
2014/02/05T03:10:00|         1906.00|        3665742452.00|         4284094.00|
2014/02/05T03:20:00|         7761.00|        7498257229.00|        17928183.00|
2014/02/05T03:30:00|         5148.00|        6367066043.00|         6150631.00|
2014/02/05T03:40:00|         1871.00|        2800838613.00|         3651024.00|
2014/02/05T03:50:00|         6272.00|        8929532068.00|        17278615.00|
2014/02/05T04:00:00|         5768.00|        1091642553.00|         3118841.00|
2014/02/05T04:10:00|         1801.00|        1631078840.00|         2115298.00|
2014/02/05T04:20:00|         5708.00|        5862014892.00|        15043499.00|
2014/02/05T04:30:00|         6108.00|        4307956317.00|         5208716.00|
2014/02/05T04:40:00|         1702.00|        1620065163.00|         2394512.00|
2014/02/05T04:50:00|         4768.00|        5080092469.00|        14051789.00|
2014/02/05T05:00:00|         6583.00|        2515417934.00|         3890446.00|
2014/02/05T05:10:00|         1613.00|        1456687394.00|         2471948.00|
2014/02/05T05:20:00|         4204.00|        6452811275.00|        15838855.00|
2014/02/05T05:30:00|         6907.00|        2354577045.00|         3009578.00|
2014/02/05T05:40:00|         1593.00|         462926632.00|         1192327.00|
2014/02/05T05:50:00|         3801.00|        7098682274.00|        15132404.00|
```

Figure 3.5: Summary of `rwcount`

rwcount

Description	Calculates volumes over time periods of equal duration
Call	`rwcount flowrecs.rw --bin-size=3600`
Parameters	**`--bin-size`** Specifies the number of seconds per bin **`--load-scheme`** Specifies how the flow volume is allocated to bins **`--skip-zeroes`** Does not print empty bins **`--start-time`** Specifies the initial time of the first bin **`--end-time`** Specifies a time in the last bin, extended to make a whole bin **`--bin-slots`** Displays timestamps as internal bin indices For additional parameters, see Table 3.13 on page 71 and Table 3.14.

3.4.1 Examining Traffic Over a Period of Time

`rwcount` is used frequently to provide graphs showing activity over long periods of time. An example considers TCP traffic reaching a targeted server. The file `in_month.rw` contains all incoming traffic reaching the address 192.168.242.209 during the period from February 5 to February 11 (a total of 375,903 flow records). The command in Example 3.12 assigns all the flows in the input file, and all the packets and bytes contained in those flows, to 30-second bins (the default bin size). All the time bins, from the bin containing the earliest flow's start time to the bin containing the latest flow's end time, will be displayed.

Example 3.12: `rwcount` Sending Results to Disk

```
<1>$ rwcount --delimited=, in_month.rw --output-path=in_month.count
```

Example 3.12 directs output to a disk file. Count data can be read by most plotting (graphing) applications; for this example, graphs are generated using the `gnuplot` utility. The resulting plot is shown in Figure 3.6. In that figure, it is impossible to discern a pattern because the default bin size of 30 seconds is too small for these data. To make the results more readable, an analyst changes the bin size to a more manageable value using `--bin-size`. Example 3.13 changes the size of the bins to an hour (3,600 seconds).

Example 3.13: `rwcount --bin-size` to Better Scope Data for Graphing

```
<1>$ rwcount in_month.rw --bin-size=3600 --output-path=in_month_h.count
```

With volumes totaled by the hour, regular traffic patterns are more visible. In Figure 3.7, these appear as a more solid wavering line, with daily peaks corresponding to somewhere between 05:00 and 09:00.

3.4.2 Characterizing Traffic by 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

Figure 3.6: Displaying `rwcount` Output Using `gnuplot`

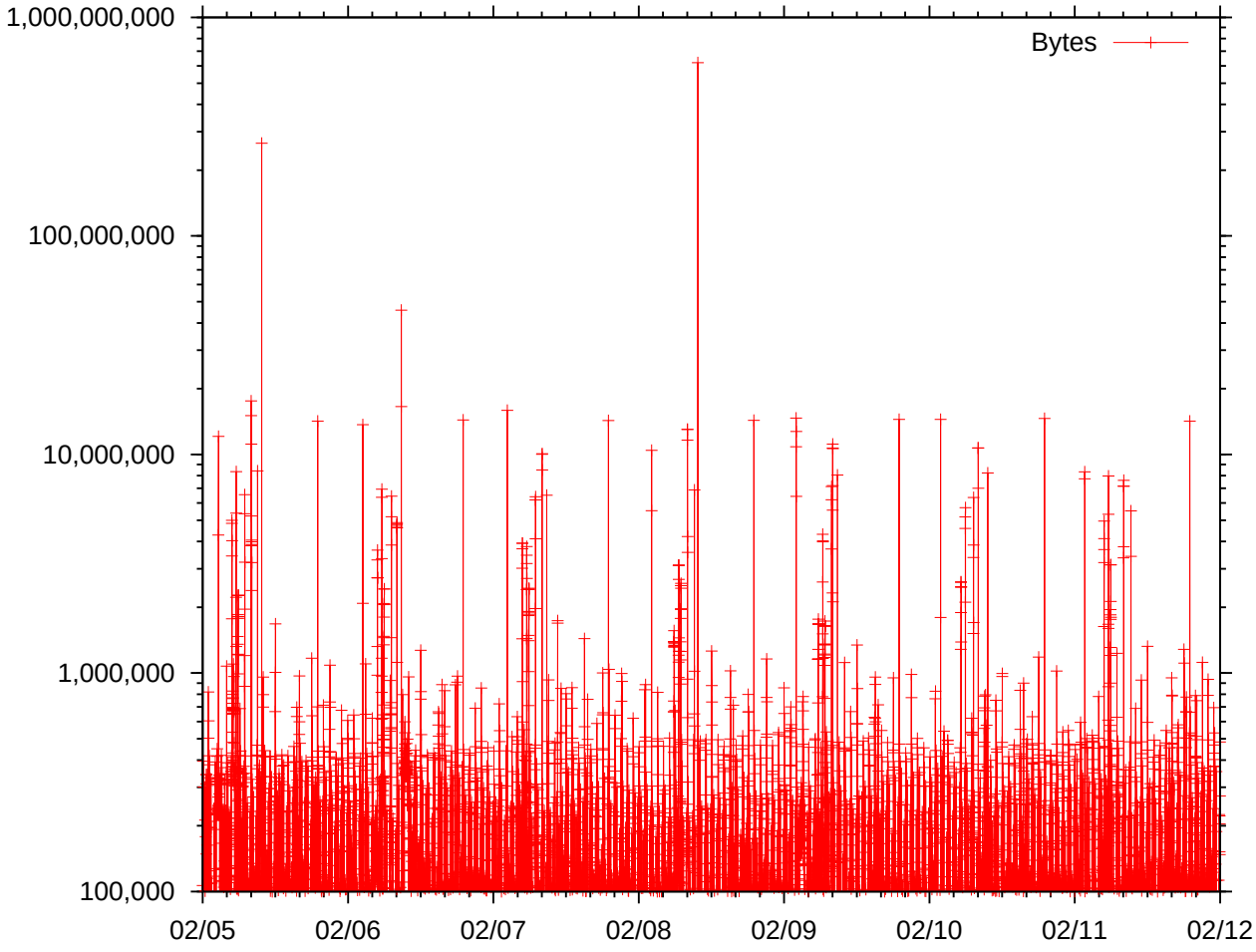
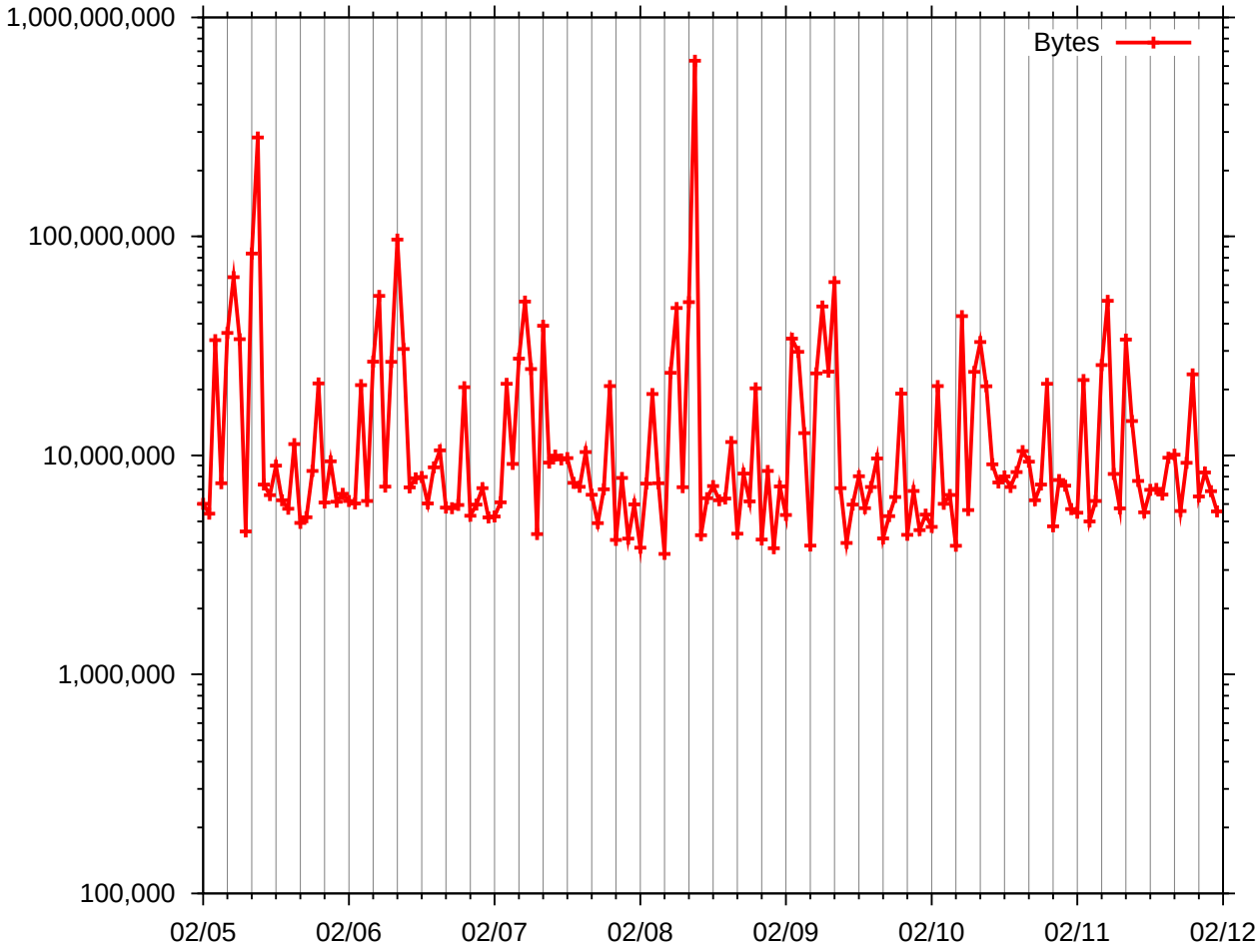
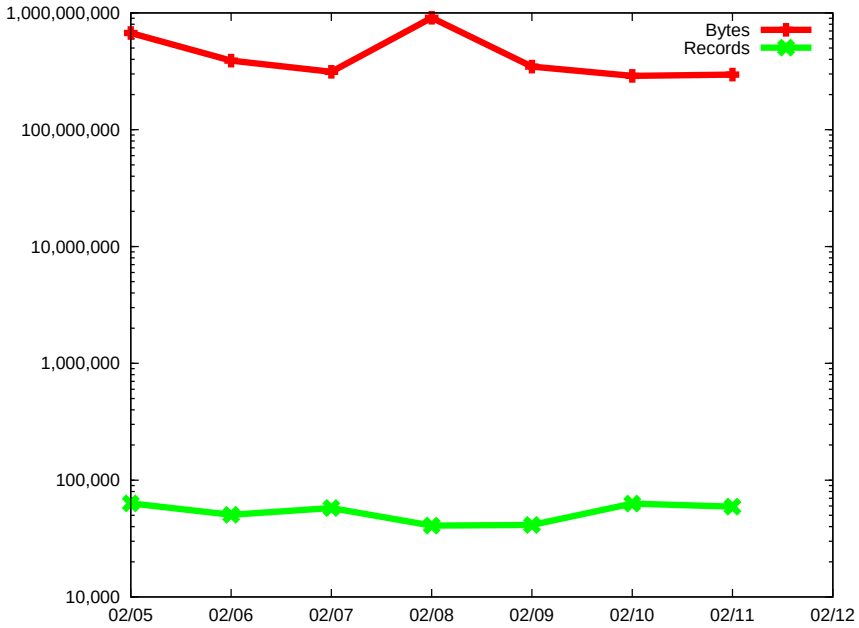


Figure 3.7: Improved `gnuplot` Output Based on a Larger Bin Size



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.

Figure 3.8: Comparison of Byte and Record Counts over Time



In Figure 3.8, a logarithmic scale enables the display of plots by byte and by packet on the same graph, using daily aggregate counts from the `in_month.rw` file. Under normal circumstances, the byte count will be several thousand times larger than the record count.

3.4.3 Changing the Format of Dates to Feed Other Tools

`rwcount` can alter its output format to accommodate different representations of time. The most important of these features are the `--timestamp-format` and `--bin-slots` parameters. `--timestamp-format` alters output to display times in a choice of formats described in Table 3.16 on page 73. The epoch format is easier to parse and sort than a conventional year/month/day format, making it useful when feeding timestamps to scripts that must sort or perform arithmetic with timestamps. The `--bin-slots` option displays timestamps as `rwcount`'s internal bin index, which is very compact but has only an ordinal relationship to specific times. Example 3.14 shows epoch time and bin slots and their relationships to normal time.

As Example 3.14 shows, the epoch values are actually the same times as in Command 1, but they are given as a single integer denoting seconds since the UNIX epoch (January 1, 1970). Also note that the epoch slots are exactly 3,600 seconds (our bin size) apart in each case. This spacing is normally expected for the conventional representation given above, but it is easier to see in this example.

`rwcount` normally starts printing at the first nonzero slot; however, to contrast flow files that start at different times, this default behavior should be overridden. To force `rwcount` to start each report at the same time, use the `--start-time` parameter as shown in Example 3.15. This parameter will force `rwcount` to start

Example 3.14: `rwcount` Alternate Date Formats

```
<1>$ rwcount in_month.rw --bin-size=3600 | head -n 4
                Date|         Records|               Bytes|           Packets|
2014/02/05T00:00:00|         1551.00|          6012000.00|          54594.00|
2014/02/05T01:00:00|         3585.33|          5422626.56|          49072.99|
2014/02/05T02:00:00|          966.67|         33596728.44|         660769.01|
<2>$ rwcount in_month.rw --bin-size=3600 --timestamp-format=epoch | head -n 4
                Date|         Records|               Bytes|           Packets|
          1391558400|         1551.00|          6012000.00|          54594.00|
          1391562000|         3585.33|          5422626.56|          49072.99|
          1391565600|          966.67|         33596728.44|         660769.01|
<3>$ rwcount in_month.rw --bin-size=3600 --timestamp-format=m/d/y | head -n 4
                Date|         Records|               Bytes|           Packets|
02/05/2014 00:00:00|         1551.00|          6012000.00|          54594.00|
02/05/2014 01:00:00|         3585.33|          5422626.56|          49072.99|
02/05/2014 02:00:00|          966.67|         33596728.44|         660769.01|
<4>$ rwcount in_month.rw --bin-size=3600 --bin-slots | head -n 4
                Date|         Records|               Bytes|           Packets|
                  48|         1551.00|          6012000.00|          54594.00|
                  49|         3585.33|          5422626.56|          49072.99|
                  50|          966.67|         33596728.44|         660769.01|
```

reporting at the specified time, regardless of the actual time of the earliest flow in the file. The value of `--start-time` may be either an integral epoch value or a conventional date in the default year/month/day format.

Example 3.15: `rwcount --start-time` to Constrain Minimum Date

```
<1>$ rwcount in_month.rw --start-time=1391554800 \
             --bin-size=3600 --timestamp-format=epoch | head -n 4
                Date|         Records|               Bytes|           Packets|
          1391554800|            0.00|                0.00|              0.00|
          1391558400|         1551.00|          6012000.00|          54594.00|
          1391562000|         3585.33|          5422626.56|          49072.99|
<2>$ rwcount in_month.rw --start-time=1391569200 \
             --bin-size=3600 --timestamp-format=epoch | head -n 4
                Date|         Records|               Bytes|           Packets|
          1391569200|         3846.00|          7464085.00|          70038.00|
          1391572800|          897.63|         36217408.79|         398225.72|
          1391576400|         2752.37|         65192444.21|         579936.28|
```

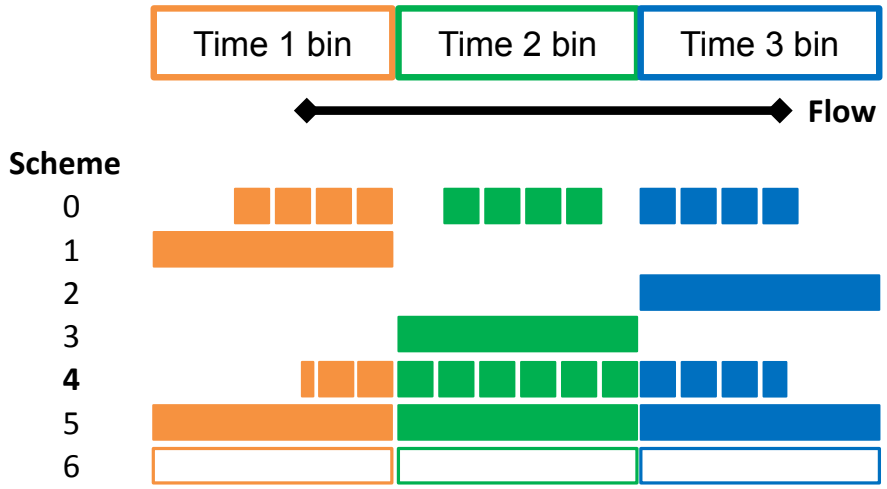
Example 3.15 shows how `--start-time` affects output. The first case starts the period before data is present; therefore, `rwcount` prints out an empty slot for the first hour. The second case starts the period after the data, and `rwcount` consequently ignores the first few slots. `--start-time` also could be used for things like one-hour bins that start on the half hour.

3.4.4 Using the --load-scheme Parameter for Different Approximations

Grouping packets into flow records results in a loss of timing information; specifically, it is not possible to tell how the packets in a flow are distributed in time. Even at the sensor, the information about the time distribution of packets in a flow is lost. Furthermore, the flow's actual time distribution of packets may not be as interesting as intentionally induced distortions of the distribution in order to emphasize certain characteristics. `rwcount` can assign one of seven time distributions of packets and bytes in the flow to allocate the volume to time bins. The choice is made by the analyst and communicated to `rwcount` with the `--load-scheme` parameter. The default scheme (4=`time-proportional`) splits the bytes, packets, and record count (1 for one flow) in all bins covered by the flow record proportionally, according to how much time the flow spent in each bin's timespan. Another scheme (0=`bin-uniform`) allocates equal parts of the volume to each of the bins containing any part of the flow's timespan. `rwcount` also can store the flow's volume in the bin containing the flow's `sTime` (1=`start-spike`), in the bin containing the flow's `eTime` (2=`end-spike`), or in the bin containing the middle millisecond of that range (3=`middle-spike`). Those five schemes share the feature that, in all of them, the record count adds up to one, and the packet and byte counts add up to counts found in the flow record. The two other schemes produce incorrect totals but are useful nevertheless. The `maximum-volume` scheme (#5) gives a worst-case reading of the volume by assigning the whole flow with all its packets and bytes to each bin covering any part of the flow's timespan. The `minimum-volume` scheme (#6) gives a best-case reading of the volume by assigning one whole flow to each related bin and assigning zero packets and bytes to those bins. Most of the time, the default scheme is the best choice.

Figure 3.9 illustrates the allocation of flows, packets, and bytes to time bins using the various load-schemes. The squares depict a fixed number of packets or bytes, and the partial squares are proportional to the complete squares. The wide, solidly filled rectangles depict entire flows, along with their packets and bytes; so these appear once in schemes 1, 2, and 3 (where one bin receives the entire flow with its packets and bytes), while these appear in every bin for scheme 5 (`maximum-volume`). The wide, hollow rectangles represent whole flows with no packets or bytes, and only appear in scheme 6 (`minimum-volume`).

Figure 3.9: `rwcount` Load-Schemes



Here are some guidelines for choosing a load-scheme. The `bin-uniform` scheme (#0) gives the average load per bin, providing smooth peaks and valleys among bins. The `start-spike` scheme (#1) emphasizes onset

or periodic behavior. The `end-spike` scheme (#2) emphasizes flow termination. The `middle-spike` scheme (# 3) emphasizes payload transfer above setup or termination. The `time-proportional` scheme (#4) gives the average load per time period, providing smooth peaks and valleys over time. The `maximum-volume` scheme (#5) provides worst-case service loading. The `minimum-volume` scheme (#6) provides best-case service loading.

3.5 Displaying Flow Records Using `rwcut`

SiLK uses binary data to implement fast access and file manipulation; however, these data cannot be read using any of the standard text-processing UNIX tools. As shown in Figure 3.10, the `rwcut` command reads flow files and produces human-readable output in a pipe-delimited tabular format.

`rwcut` reads flow files and formats their flows as readable text. Using the `--fields` parameter, SiLK data fields can be selected, reordered, formatted as text, and separated from each other in different ways. The values for the `sIP`, `dIP`, `nhIP`, `sTime`, `eTime`, `sensor`, `flags`, `initialFlags`, and `sessionFlags` fields can be displayed in multiple formats.

Figure 3.10: Summary of `rwcut`

rwcut

Description	Reads SiLK flow data and displays it as text
Call	`rwcut flowrecs.rw --fields=1-5,sTime`
Parameters	Choose one or none: `--fields` Specifies which fields to display (default is 1–12) `--all-fields` Displays all fields Options: `--start-rec-num` Specifies record offset of first record from start of file `--end-rec-num` Specifies record offset of last record from start of file `--tail-recs` Specifies record offset of first record from end of file (cannot combine with `--start-rec-num` or `--end-rec-num`) `--num-recs` Specifies maximum number of output records For additional parameters, see Table 3.13 on page 71 and Table 3.14.

`rwcut` is invoked in two ways, either by reading a file or by connecting it with another SiLK tool (often `rwfilter` or `rwsort`) via a pipe. When reading a file, just specify the filename in the command line, as shown in Example 3.16.

To use `rwcut` with `rwfilter`, connect them together with a pipe, as illustrated in Example 3.17.

3.5.1 Pausing Results with Pagination

When output is sent to a terminal, `rwcut` (and other text-outputting tools) will 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 `PAGER`. If `SILK_PAGER` contains the empty string, as shown

Example 3.16: `rwcut` for Displaying the Contents of a File

```
<1>$ rwcut fastfile.rw --fields=1-5,packets --num-recs=10
            sIP|            dIP|sPort|dPort|pro|   packets|
    192.0.2.226| 192.168.200.39|11229|51015|  6|        21|
    192.0.2.226| 192.168.200.39|34075|44230|  6|        21|
    192.0.2.226| 192.168.200.39|23347|33503|  6|        21|
   203.0.113.15|192.168.111.219|59475|57359|  6|       153|
   203.0.113.15|192.168.111.219|   21|57358|  6|        38|
   203.0.113.43|  192.168.60.88|33718|54823|  6|         6|
    203.0.113.7| 192.168.187.44|  389| 4785|  6|        43|
     192.0.2.78|192.168.214.241|   21|49913|  6|        17|
 198.51.100.242| 192.168.224.53| 1674|54264|  6|        16|
    192.0.2.117| 192.168.28.178|   25|58986|  6|        23|
```

Example 3.17: `rwcut` Used with `rwfilter`

```
<1>$ rwfilter slowfile.rw --saddress=x.x.x.32 --max-pass-records=4 \
              --pass=stdout \
      | rwcut --fields=1-5,packets
            sIP|            dIP|sPort|dPort|pro|   packets|
   172.31.68.32|192.168.176.200|  123|  123| 17|        30|
   172.31.41.32|  192.168.95.33|    0| 2048|  1|       598|
    172.31.4.32|  192.168.250.6|  123|  123| 17|        29|
  172.31.102.32|192.168.176.200|    0| 2048|  1|       161|
```

in Example 3.18 for the Bash shell, no paging will be performed. The paging program can be specified for an invocation of a tool by using its `--pager` parameter, as shown in Example 3.19.

Example 3.18: `SILK_PAGER` with the Empty String to Disable Paging

```
<1>$ export SILK_PAGER=
```

Example 3.19: `rwcut --pager` to Disable Paging

```
<1>$ rwfilter ... | rwcut --field=sTime --pager=
```

3.5.2 Selecting Fields to Display

The `--fields` parameter provides a means for `rwcut` to select fields to appear in output. Other SiLK tools, such as `rwstats`, `rwsort`, and `rwuniq` use the `--fields` parameter to specify the composition of the key. With `rwcut`, that parameter controls which stored fields and derived fields from each flow record are displayed. The argument to this parameter is a list of field number ranges (which may be just a single number), field names, or a mixture of the two.

Table 3.11 shows these field numbers and names. In this table, the "Field Name" column may hold several names separated by commas; these names are equivalent. Unlike option names (such as `--fields` or `--values`), the field names themselves (e.g., sIP, sPort) are case-insensitive; so although the table shows the preferred capitalization, any capitalization is acceptable. In some cases, a field name may not have a corresponding field number, indicating that the name must be used when this field is desired. If a given field refers to a name defined in a prefix map, a plug-in, or a PySiLK file, that file must be loaded (via the `--pmap-file`, `--plugin`, or `--python-file` parameter) prior to specifying the `--fields` parameter. (See Section 4.7 for more information on prefix maps.)

3.5.3 Rearranging Fields for Clarity

The `--fields` parameter can also be used to rearrange fields. Because the order in which field names and numbers are specified is significant, `--fields=1,2,3` will result in a display that is different from `--fields=3,2,1`. Example 3.20 displays the output fields in this order: source IP address, source port, start time, destination IP address.

Example 3.20: `rwcut --fields` to Rearrange Output

```
<1>$ rwfilter fastfile.rw --protocol=6 --max-pass-records=4 --pass=stdout \
      | rwcut --fields=1,3,sTime,2
            sIP|sPort|                  sTime|            dIP|
 198.51.100.121|11229|2014/02/05T00:00:00.120|  192.168.62.82|
 198.51.100.121|34075|2014/02/05T00:00:00.982|  192.168.62.82|
 198.51.100.121|23347|2014/02/05T00:00:01.762|  192.168.62.82|
  203.0.113.177|59475|2014/02/05T00:00:00.895|192.168.241.110|
```

Table 3.11: Arguments for the `--fields` Parameter

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

3.5.4 Selecting Fields for Performance

In general, the SiLK tools provide sufficient filtering and summarizing facilities to make performance scripting rare. However, given the volume of data that can be processed, it is worth considering performance constraints between `rwcut` and scripts. Left to its default parameters, `rwcut` prints a lot of characters per record; with enough records, script execution can be quite slow.

In Example 3.21, Command 1 pulls a fairly large file. Command 2 uses `rwcut` and the UNIX line-count command `wc -l` to count the number of records in the file. The UNIX shell reserved word `time` is used to determine how long `rwcut` and `wc` run. In the output, the first line is the record count, and the three lines after the blank line are the result of `time`. The performance of the command is mostly reflected in the sum of the user time and the system time, giving the number of CPU seconds dedicated to the task. The real time (that is, what humans would observe on a wall clock) will include any input-output (I/O) wait time, but will also include time due to unrelated processes. The real time can be less than the sum of user and system times, because multiple processors or cores will result in some overlap of user and system times.

Example 3.21: `rwcut` Performance with Default `--fields`

```
<1>$ rwfilter --start-date=2014/02/05T00 \
              --protocol=6 --pass=tmp.rw
<2>$ time rwcut tmp.rw --no-titles | wc -l
7597736

real    0m59.157s
user    0m13.982s
sys     0m55.180s
```

Compare the sum of user and system times with the results shown in Example 3.22, which eliminates all the fields except `protocol`. The result is an order of magnitude faster.

Example 3.22: `rwcut --fields` to Improve Efficiency

```
<1>$ time rwcut --field=protocol --no-titles tmp.rw | wc -l
7597736

real    3m19.018s
user    0m2.883s
sys     0m2.047s
```

`rwcut` will be a tiny bit faster using the `--no-columns` or `--delimited` parameter (since the process won't have to pad fields and there will be less text to output).

3.5.5 Modifying Field Formatting for Clarity

Since `rwcut` is the fundamental command for displaying SiLK flow records, it includes several features for reformatting and modifying output. In general, `rwcut`'s features are minimal and focus on the most relevant high-volume tasks, specifically generating data that can then be read easily by other scripting tools.

Changing Field Formats for ICMP

ICMP types and codes are stored in the destination port field for most sensors. Normally, this storage results in a value equivalent to $(type \times 256) + code$ being stored in the `dPort` field, as shown in Example 3.23.

Example 3.23: `rwcut` ICMP Type and Code as `dPort`

```
<1>$ rwfilter in_month.rw --protocol=1 --max-pass-records=5 --pass=stdout \
      | rwcut --fields=1-5,packets,bytes
            sIP|            dIP|sPort|dPort|pro|   packets|     bytes|
    192.0.2.92|192.168.242.209|    0|  769|  1|         1|        56|
    192.0.2.27|192.168.242.209|    0|  778|  1|         1|        56|
   192.0.2.160|192.168.242.209|    0|  771|  1|         1|       103|
198.51.100.254|192.168.242.209|    0|  772|  1|         2|      1152|
198.51.100.139|192.168.242.209|    0| 2048|  1|         1|        32|
```

`rwcut` includes field names that decode the `dPort` input field into separate `iType` and `iCode` output fields. Example 3.24 shows the same data reformatted using the ICMP type and code field names. Command 1 uses the older field name `icmpTypeCode` (deprecated as of SiLK Version 3.8.1), which actually displays two output columns: `iTy` and `iCo`. Command 2 uses the newer (as of SiLK Version 3.8.1) field names, `iType` and `iCode`, permitting the analyst to display the ICMP type without the ICMP code, change the order of fields, or interpose other fields between them.

Example 3.24: `rwcut` Using ICMP Type and Code Fields

```
<1>$ rwfilter in_month.rw --protocol=1 --max-pass-records=5 --pass=stdout \
      | rwcut --fields=1-2,icmpTypeCode,5,packets,bytes
            sIP|            dIP|iTy|iCo|pro|   packets|     bytes|
    192.0.2.92|192.168.242.209|  3|  1|  1|         1|        56|
    192.0.2.27|192.168.242.209|  3| 10|  1|         1|        56|
   192.0.2.160|192.168.242.209|  3|  3|  1|         1|       103|
198.51.100.254|192.168.242.209|  3|  4|  1|         2|      1152|
198.51.100.139|192.168.242.209|  8|  0|  1|         1|        32|
<2>$ rwfilter in_month.rw --protocol=1 --max-pass-records=5 --pass=stdout \
      | rwcut --fields=1-2,iType,iCode,5,packets,bytes
            sIP|            dIP|iTy|iCo|pro|   packets|     bytes|
    192.0.2.92|192.168.242.209|  3|  1|  1|         1|        56|
    192.0.2.27|192.168.242.209|  3| 10|  1|         1|        56|
   192.0.2.160|192.168.242.209|  3|  3|  1|         1|       103|
198.51.100.254|192.168.242.209|  3|  4|  1|         2|      1152|
198.51.100.139|192.168.242.209|  8|  0|  1|         1|        32|
```

Changing Field Separators to Support Later Processing

The `--delimited` parameter allows changing the separator from a pipe (`|`) to any other character. Example 3.25 shows this replacement. When using that parameter, spacing is removed as well. This is particularly useful with `--delimited=','` which produces comma-separated-value output for easy import into Excel and other tools. The `--delimited` parameter doesn't allow multi-character separators to be defined, as it uses

only the first character of the argument. To produce a separator consisting of a comma and a space, use the `--column-separator` parameter, which changes the separator without affecting the spacing the way that `--delimited` does.[12] The `--no-columns` parameter suppresses spacing between columns without affecting the separator. The `--ip-format` parameter allows `rwcut` to display IP addresses in formats other than the default IPv4 dotted-decimal notation or IPv6 canonical notation, such as integers or zero-padded. The `--no-titles` parameter (see Example 3.26) suppresses the column headings, which can be useful when doing further processing with text-based tools.

Example 3.25: `rwcut --delimited` to Change the Delimiter

```
<1>$ rwcut fastfile.rw --fields=1-3 --ip-format=decimal --delimited=, \
           --num-recs=5
sIP,dIP,sPort
3325256825,3232251474,11229
3325256825,3232251474,34075
3325256825,3232251474,23347
3405803953,3232297326,59475
3405803953,3232297326,21
```

Example 3.26: `rwcut --no-titles` to Suppress Column Headings in Output

```
<1>$ rwcut slowfile.rw --no-titles --num-recs=3 --fields=1-5
    192.0.2.195| 192.168.52.160|43152|   21|  6|
  172.31.63.231| 192.168.123.10| 1935|50047|  6|
  172.31.63.231| 192.168.123.10| 1935|50045|  6|
```

3.5.6 Selecting Records to Display

`rwcut` has four record selection parameters: `--num-recs`, `--start-rec-num`, `--end-rec-num`, and `--tail-recs`. `--num-recs` limits the number of records output, as shown in Example 3.27.

Example 3.27: `rwcut --num-recs` to Constrain Output

```
<1>$ rwfilter slowfile.rw --protocol=6 --pass=stdout \
      | rwcut --fields=1-5 --num-recs=3
            sIP|            dIP|sPort|dPort|pro|
    192.0.2.195| 192.168.52.160|43152|   21|  6|
  172.31.63.231| 192.168.123.10| 1935|50047|  6|
  172.31.63.231| 192.168.123.10| 1935|50045|  6|
```

`--num-recs` forces a maximum number of lines of output. For example, using `--num-recs=100`, up to 100 records will be shown regardless of pipes or redirection. As shown in Example 3.28, the line count of 101 includes the title line. To eliminate that line, use the `--no-titles` option (see Example 3.26).

[12] Or pipe through the UNIX command `sed -e 's/,/, /g'` to add a space after each comma.

Example 3.28: `rwcut --num-recs` and Title Line

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwcut --fields=1-5 --num-recs=100 | wc -l
101
```

`--num-recs` will print out records until it reaches the specified number of records or there are no more records to print. If multiple input files are specified on the `rwcut` command line, `--num-recs` will read records from the input files in the order specified until `--num-recs` is satisfied and then will read no further. One file may be read only partially, and some files may be completely unread. The `--start-rec-num` and `--end-rec-num` options are used to specify the record number at which to start and stop printing. The first record is record number one.

Example 3.29 prints out the records in the `fastfile.rw` file starting at record two.

Example 3.29: `rwcut --start-rec-num` to Select Records to Display

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwcut --fields=1-5 --start-rec-num=2 --num-recs=10
            sIP|            dIP|sPort|dPort|pro|
 198.51.100.121|  192.168.62.82|34075|44230|  6|
 198.51.100.121|  192.168.62.82|23347|33503|  6|
  203.0.113.177|192.168.241.110|59475|57359|  6|
  203.0.113.177|192.168.241.110|   21|57358|  6|
   203.0.113.59| 192.168.17.179|33718|54823|  6|
    192.0.2.184|192.168.194.196|  389| 4785|  6|
  203.0.113.106| 192.168.251.92|   21|49913|  6|
 198.51.100.149| 192.168.175.58| 1674|54264|  6|
     192.0.2.76| 192.168.212.79|   25|58986|  6|
 198.51.100.149| 192.168.175.58| 1674|54266|  6|
```

If `--end-rec-num` and `--num-recs` are both specified, output will end as soon as one of those conditions is satisfied as shown in Example 3.30.

Example 3.30: `rwcut --start-rec-num`, `--end-rec-num`, and `--num-recs` Combined

```
<1>$ rwfilter fastfile.rw --protocol=6 --pass=stdout \
      | rwcut --fields=1-5 --start-rec-num=2 --end-rec-num=8 --num-recs=99
            sIP|            dIP|sPort|dPort|pro|
 198.51.100.121|  192.168.62.82|34075|44230|  6|
 198.51.100.121|  192.168.62.82|23347|33503|  6|
  203.0.113.177|192.168.241.110|59475|57359|  6|
  203.0.113.177|192.168.241.110|   21|57358|  6|
   203.0.113.59| 192.168.17.179|33718|54823|  6|
    192.0.2.184|192.168.194.196|  389| 4785|  6|
  203.0.113.106| 192.168.251.92|   21|49913|  6|
```

3.6 Sorting Flow Records with `rwsort`

`rwsort` is a high-speed sorting tool for SiLK flow records. `rwsort` reads SiLK flow records from its input files and outputs the same records in a user-specified order. The output records, just like the input records, are 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. Figure 3.11 provides a brief summary of this tool.

Figure 3.11: Summary of `rwsort`

rwsort

Description	Sorts SiLK flow records using key field(s)
Call	`rwsort unsorted1.rw unsorted2.rw --fields=1,3` `--output-path=sorted.rw`
Parameters	**`--fields`** Specifies key fields for sorting (required) **`--presorted-input`** Specifies only merging of already sorted input files **`--reverse`** Specifies sort in descending order **`--temp-directory`** Specifies location of high-speed storage for temp files **`--sort-buffer-size`** Specifies the in-memory sort buffer (2 GB default) For additional parameters, see Table 3.13 on page 71 and Table 3.14.

The order of values in the argument to the `--fields` parameter to `rwsort` indicates the sort's precedence for fields. 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 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 several 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.

In cases where `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.

3.6.1 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, take a sample of a particular protocol, use `rwsort --fields=sTime`, and then convert the results to ASCII with `rwcut`. To produce byte and packet fields only, try `rwcut` with `--fields=bytes` and `--fields=packets`, and 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? Do any bytes-per-packet ratios fall outside the baseline?
2. 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 understanding of network behavior.

3.7 Counting Flows with `rwuniq`

`rwuniq`, summarized in Figure 3.12, is a general-purpose counting tool: It provides counts of the records, bytes, and packets for any combination of fields, including binning by time intervals. 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`.

Figure 3.12: Summary of `rwuniq`

rwuniq

Description	Counts records per combination of multiple-field keys
Call	`rwuniq filterfile.rw --fields=1-5,sensor --values=Records`
Parameters	`--fields` Specifies fields to use as key (required) `--values` Specifies aggregate counts (default: Records) `--bin-time` Establishes bin size for time-oriented bins `--presorted-input` Reduces memory requirements for presorted records `--sort-output` Sorts results by key, as specified in the `--fields` parameter For options to filter output rows, see Table 3.12. For additional parameters, see Table 3.13 on page 71 and Table 3.14.

Table 3.12: Output-Filtering Options for `rwuniq`

Parameter	Description
`--bytes`	Only output rows whose aggregate byte counts are in the specified range
`--packets`	Only output rows whose aggregate packet counts are in the specified range
`--flows`	Only output rows whose aggregate flow (record) counts are in the specified range
`--sip-distinct`	Only output rows whose counts of distinct (unique) source IP addresses are in the specified range
`--dip-distinct`	Only output rows whose counts of distinct (unique) destination IP addresses are in the specified range

Example 3.31 shows a count on source IP addresses (field 1). The count shown in Example 3.31 is a count of individual flow records.

Example 3.31: `rwuniq` for Counting in Terms of a Single Field

```
<1>$ rwuniq fastfile.rw --field=sIP | head -n 10
             sIP|   Records|
 172.30.138.196|         5|
  172.30.21.175|         9|
  172.30.172.55|         2|
 172.30.206.153|         1|
 172.29.120.119|         7|
  172.29.215.89|         1|
   172.31.6.207|         5|
 172.30.193.137|         1|
 172.29.123.130|         4|
```

3.7.1 Using Thresholds with `rwuniq` to Profile a Slice of Flows

`rwuniq` provides a capability to set thresholds on counts. For example, to show only those source IP addresses with 200 flow records or more, use the `--flows` parameter as shown in Example 3.32. In addition to providing counts of flow records, `rwuniq` can count bytes and packets through the Bytes and Packets values in the `--values` parameter, as shown in Example 3.33.

Example 3.32: `rwuniq --flows` for Constraining Counts to a Threshold

```
<1>$ rwuniq in_month.rw --field=sIP --value=Flows --flows=200- | head -n 10
             sIP|   Records|
  172.31.27.146|       249|
  172.31.83.171|       216|
 172.31.177.128|       646|
   172.31.96.95|       284|
    192.0.2.220|       220|
   172.31.219.3|       226|
  172.31.239.58|       258|
  198.51.100.99|       327|
 172.31.243.204|       203|
```

The `--bytes`, `--packets`, and `--flows` parameters are all threshold operators for filtering output rows. Without a specified range (such as `--flows=200-2000`), the parameter simply adds the named aggregate count to the list of aggregates started by the `--values` parameter; it is preferable to use the `--values` parameter for this purpose since that parameter provides greater control over the order in which the aggregates are displayed. A range may be specified in three ways: 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, or with a low bound alone (e.g., 200) which, unlike `rwfilter` partitioning values, still denotes a range with no upper bound, not just a single value. The last form is discouraged, since it is misinterpreted easily. When multiple threshold parameters are specified, `rwuniq` will print all records that meet all the threshold criteria, as shown in Example 3.34.

Example 3.33: `rwuniq --bytes` and `--packets` with Minimum Flow Threshold

```
<1>$ rwuniq in_month.rw --field=sIP --values=Bytes,Packets,Flows --flows=2000-
            sIP|               Bytes|         Packets|   Records|
      192.0.2.6|            14955948|          128191|     22563|
 198.51.100.123|           313208842|         1692517|      5083|
  203.0.113.240|            45685747|           63650|      4202|
 172.31.102.236|             5626076|           72300|      7609|
  198.51.100.18|            92736014|         1065135|    163370|
   172.31.47.56|             7053117|           74897|      6083|
  203.0.113.189|            20888986|          326759|     22007|
   203.0.113.68|           516609610|         3522213|     13069|
  198.51.100.44|             1598689|           23084|      2397|
```

Example 3.34: `rwuniq --flows` and `--packets` to Constrain Flow and Packet Counts

```
<1>$ rwuniq in_month.rw --field=sIP --values=Bytes,Packets,Flows \
            --flows=2000- --packets=100000-
            sIP|               Bytes|         Packets|   Records|
      192.0.2.6|            14955948|          128191|     22563|
 198.51.100.123|           313208842|         1692517|      5083|
  198.51.100.18|            92736014|         1065135|    163370|
  203.0.113.189|            20888986|          326759|     22007|
   203.0.113.68|           516609610|         3522213|     13069|
```

3.7.2 Counting IPv6 Flows

`rwuniq` automatically adjusts to process IPv6 flow records if they are supplied as input. No specific parameter is needed to identify these flow records, as shown in Example 3.35. This example uses `rwfilter` to isolate IPv6 "Packet Too Big" flow records (ICMPv6 Type 2) and then uses `rwuniq` to profile how often each host sends these and to how many destinations. These flows are used for Path Maximum Transmission Unit (PMTU) discovery, an optimization of packet sizing within IPv6 to prevent the need for frequent packet fragmentation and reassembly. A low number of such flow records is considered acceptable. If a source IP address has a high count, that host is throttling back network connections for communicating hosts.

Example 3.35: Using `rwuniq` to Detect IPv6 PMTU Throttling

```
<1>$ rwfilter --ip-version=6 --icmp-type=2 --pass=stdout \
      | rwuniq --fields=sIP --values=Flows,Distinct:dIP --flows=2-
                                    sIP|   Records|dIP-Distin|
 2001:db8:0:320a:9ce3:a2ff:ae0:e169|         5|         2|
            2001:db8:0:3e00::2e28:0|         2|         2|
           2001:db8:a401:fe00::51f6|         8|         1|
                2001:db8:0:64b2::a5|         2|         1|
```

3.7.3 Using Compound Keys with `rwuniq` to Profile Selected Cases

In addition to the simple counting shown above, `rwuniq` can count on combinations of fields. To use a compound key, specify it using a comma-separated list of values or ranges in `rwuniq`'s `--fields` parameter. Keys can be manipulated as in `rwcut`, so `--fields=3,1` is a different key than `--fields=1,3`. In Example 3.36, `--fields` is used to identify communication between clients and specific services only when the number of flows for the key exceeds a threshold.

Example 3.36: `rwuniq --fields` to Count with Respect to Combinations of Fields

```
<1>$ rwfilter in_month.rw --protocol=6 --pass=stdout \
      | rwuniq --fields=sIP,sPort --value=Flows --flows=20- \
      | head -n 11
            sIP|sPort|   Records|
 172.31.102.236|40553|        24|
     192.0.2.37|   80|        21|
  172.31.46.127|54168|        21|
 172.31.102.236|41622|        20|
 198.51.100.123|   22|      5083|
 172.31.102.236|40546|        20|
    192.0.2.104|39135|        59|
 172.31.188.210|13904|        24|
 198.51.100.121|   22|       713|
 198.51.100.144|20806|        21|
```

In Example 3.36, incoming traffic is used to identify those source IP addresses with the highest number of flow records connecting to specific TCP ports.

3.7.4 Using rwuniq to Isolate Behavior

rwuniq can profile flow records for a variety of behaviors, by first filtering for the behavior of interest and then using rwuniq to count the records showing that behavior. This can be useful in understanding hosts that use or provide a variety of services. Example 3.37 shows how to generate data that compare hosts showing email and non-email behavior among a group of flow records. Command 1 first isolates the set of hosts of interest (using an IP-set file created previously), divides their records into mail and non-mail behaviors (by protocol and port), and finally counts the mail behavior into a file, sorted by source address. Command 2 counts the non-mail 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. Command 3 merges the two count files by source address and then sorts them by number of mail flows with the results shown. Hosts with high counts in both columns should be either workstations or gateways. Hosts with high counts in email and low counts in non-email should be email servers.[13] For more complex summaries of behavior, use the bag utilities as described in Section 4.5.

Example 3.37: Using rwuniq to Isolate Email and Non-Email Behavior

```
<1>$ rwfilter in_month.rw --sipset=interest.set --pass=stdout \
      | rwfilter stdin --protocol=6 --aport=25 \
                  --fail=more-nomail.rw --pass=stdout \
      | rwuniq --field=sIP --no-titles --ip-format=zero-padded \
               --sort-output --output-path=more-mail-saddr.txt
<2>$ rwuniq more-nomail.rw --field=sIP --no-titles --ip-format=zero-padded \
            --sort-output --output-path=more-nomail-saddr.txt
<3>$ echo '             sIP|      mail||  not mail|' \
      ; join more-mail-saddr.txt more-nomail-saddr.txt \
      | sort -t'|' -nrk2,2 \
      | head -n 5
            sIP|      mail||  not mail|
172.031.147.234|        32||        52|
172.031.132.041|        20||        19|
172.031.164.174|        19||        19|
192.000.002.104|        12||       454|
192.000.002.101|        12||       148|
```

3.8 Comparing rwstats to rwuniq

rwstats in top or bottom mode and rwuniq have much in common, especially since SiLK version 3.0.0. Many times, an analyst could accomplish a task 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 aggregates 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

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

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 aggregate counts in the `--values` parameter, the first aggregate count is the primary aggregate count. `rwstats` can apply a threshold only to the primary aggregate count, while `rwuniq` can apply thresholds to any or several aggregate counts.

For display of all bins, `rwuniq` is easiest to use, but a similar result can be obtained with `rwstats --threshold=1`. `rwstats` will run slower though, since it must sort the bins by their aggregate values.

`rwstats` always sorts bins by their primary aggregate 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 aggregate count is not bytes, packets, or records.

`rwuniq` has two aggregate 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.

3.9 Features Common to Several Commands

Many of the SiLK tools share features such as using common parameters, providing help, handling the two versions of IP addresses, and controlling the overwriting of existing output files.

3.9.1 Parameters Common to Several Commands

Many options apply to several of the SiLK tools, as shown in Table 3.13.

Table 3.14 lists the same options as in Table 3.13 and provides descriptions of the options. Three of the options described accept a small number of fixed values; acceptable values for `--ip-format` are listed and described in Table 3.15, values for `--timestamp-format` are described in Table 3.16, and values for `--ipv6-policy` are described in Table 3.17 on page 76.

3.9.2 Getting Tool Help

All SiLK tools include a help screen that provides a summary of command information. The help screen can be invoked by using the `--help` parameter with the command.

SiLK is distributed with conventional UNIX manual pages and *The SiLK Reference Guide*, both of which explain all the parameters and functionality of each tool in the suite.

All SiLK tools also have a `--version` parameter (as shown in Command 2 of Example 3.38) that identifies the version installed. Since the suite is still being extended and evolved, this version information may be quite important.

Table 3.13: Common Parameters in Essential SiLK Tools

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 3.14: Parameters Common to Several Commands

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 4.7)
`--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 3.17)
`--ip-format`	Chooses the format of IP addresses in output (see Table 3.15)
`--timestamp-format`	Chooses the format and/or timezone of timestamps in output (see Table 3.16)
`--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 screenful 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 3.15: `--ip-format` Values

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 3.16: `--timestamp-format` Values

Value	Description
`default`	Formats timestamps as *YYYY*/*MM*/*DD*T*hh*:*mm*:*ss*
`iso`	Formats timestamps as *YYYY*-*MM*-*DD* *hh*:*mm*:*ss*
`m/d/y`	Formats timestamps as *MM*/*DD*/*YYYY* *hh*:*mm*:*ss*
`epoch`	Formats timestamps as an integer of the number of seconds since 1970/01/01 00:00:00 UTC (UNIX epoch)
`utc`	Specifies timezone to use Coordinated Universal Time (UTC)
`local`	Specifies timezone to use the `TZ` environment variable or the system timezone

Example 3.38: Using `--help` and `--version`

```
<1>$ rwsetmember --help
rwsetmember [SWITCHES] WILDCARD_IP INPUT_SET [INPUT_SET...]
        Determine existence of IP address(es) in one or more IPset files.
        By default, print names of INPUT_SETs that contain WILDCARD_IP.

SWITCHES:
--help No Arg. Print this usage output and exit. Def. No
--version No Arg. Print this program's version and exit. Def. No
--count No Arg. Print count of matches along with filenames
--quiet No Arg. No output, only set exit status

<2>$ rwset --version
rwset: part of SiLK 3.8.2; configuration settings:
    * Root of packed data tree:         /data
    * Packing logic:                    Run-time plug-in
    * Timezone support:                 UTC
    * Available compression methods:    none [default], zlib, lzo1x
    * IPv6 network connections:         yes
    * IPv6 flow record support:         yes
    * IPFIX/NetFlow9 collection:        ipfix,netflow9
    * Transport encryption:             GnuTLS
    * PySiLK support:                   /usr/lib64/python/site-packages
    * Enable assert():                  no
Copyright (C) 2001-2014 by Carnegie Mellon University
GNU General Public License (GPL) Rights pursuant to Version 2, June 1991.
Some included library code covered by LGPL 2.1; see source for details.
Government Purpose License Rights (GPLR) pursuant to DFARS 252.227-7013.
Send bug reports, feature requests, and comments to netsa-help@cert.org.
```

3.9.3 Overwriting Output Files

SiLK commands, by default, do not permit their output files to be overwritten. This design choice was made to guard against the careless loss of results by re-execution of a previous command or a command being incompletely edited. If the analyst needs to overwrite previous files, there are three basic options:

- Manually remove the output file before running the SiLK command, as shown in Example 3.39.
- Set the `SILK_CLOBBER` environment variable to a non-empty, nonzero value (such as `1` or `YES`) at the start of the command line for the SiLK tool, for example

  ```
  SILK_CLOBBER=1 rwsettool --union new.set cuml.set --output-path=temp.set
  ```

  Note that there is no separator (semicolon or line break) between the variable assignment and the command: Adding one will defeat the intended purpose.
- Globally set this environment variable, via

  ```
  export SILK_CLOBBER=1
  ```

  This is not recommended, because it globally allows all SiLK commands to overwrite output files until the environment variable is cleared by the `unset` command or exported with an empty or zero value.

Example 3.39: Removing Previous Output

```
<1>$ rm -f temp.set
<2>$ rwsettool --union new.set cuml.set --output-path=temp.set
<3>$ mv temp.set cuml.set
```

3.9.4 IPv6 Address Policy

If the SiLK suite is configured to handle IPv6 records, the processing of all addresses, whether IPv4 or IPv6, is governed by the address policy. This policy affects a variety of commands, including `rwcut`, `rwuniq`, `rwstats`, `rwset`, and `rwbag` but does not affect `rwfilter` (which uses the `--ip-version` parameter) or commands that process based on their input, such as `rwtuc`, `rwptoflow`, `rwsetbuild`, and `rwpmapbuild`. There are five options for this policy, as listed in Table 3.17 and illustrated in Example 3.40.

SiLK flow records have an IP version associated with them. That is, the flow record is identified as IPv4 or IPv6 rather than the individual addresses within the record. Each record contains three IP addresses: source (sIP), destination (dIP), and next-hop (nhIP). The `nhIP` field is given the zero address by dedicated sensors such as `yaf`; only routers that act as sensors provide an actual next-hop address in this field. Some SiLK commands (e.g., `rwgroup`) place addresses or other information in this field.

When an IPv4 address appears in an IPv6 record, it is stored as an IPv4-mapped address, in the form `::FFFF:`$w.x.y.z$. How an IPv4-mapped address appears in output depends on the `--ipv6-policy` option, particularly when it is set to `asv4`.

In Example 3.40, only the first record is an IPv4 record. The SiLK tool that created the record was presented with two IPv4 addresses, so it created an IPv4 record. The second and fifth records were presented to the

Table 3.17: `--ipv6-policy` Values

Value	Description
`ignore`	Processes IPv4 input records but not IPv6 ones. This choice displays addresses in 15-character columns instead of 39-character columns because all the addresses are IPv4 addresses.
`asv4`	Processes IPv4 records and some IPv6 records. Converts an IPv6 record to an IPv4 record if the source and destination addresses are both IPv4-mapped addresses and the next-hop address is either zero or an IPv4-mapped address. Ignores other IPv6 records. This choice displays addresses in 15-character columns.
`mix`	Processes both IPv4 and IPv6 records. Treats all addresses as having the same version as the record that contains them. This choice displays addresses in 39-character columns, since IPv6 records may be present.
`force`	Processes both IPv4 and IPv6 records. Converts IPv4 records to IPv6 records, converting IPv4 addresses to IPv4-mapped IPv6 addresses. This choice displays addresses in 39-character columns.
`only`	Processes IPv6 input records but not IPv4 ones. This choice displays addresses in 39-character columns.

tool with an IPv6 address and an IPv4 address, creating an IPv6 record with the IPv4 addresses stored as IPv4-mapped IPv6 addresses.

The output in Commands 1, 3, and 5 of Example 3.40 have had 15 spaces removed from the two address columns to allow the output to fit within page margins.

- Command 1 in this example shows the default for most configurations: It permits a mixture of IPv4 and IPv6 formatted records. Addresses that were Version 4 before they were stored in IPv6 records are now stored as IPv4-mapped IPv6 addresses and appear as IPv6 addresses with the `mix` policy.
- Command 2 shows the policy of ignoring IPv6 format records, resulting in only one record output and in formatting space reduced to the needs of IPv4 addresses.
- Command 3 shows the policy of processing only IPv6 records, which again reduces the number of records displayed and also expands the formatting space to that required by IPv6 addresses.
- Command 4 coerces IPv6 addresses to IPv4 format if possible, which processes more records than Command 2 but fewer than all the records, and reduces the formatted width to the space required by IPv4 records. It is possible to format an IPv6 address as IPv4 if the address is in the IPv4-mapped or IPv4-compatible netblocks (see Table 1.1 on page 9).
- Command 5 forces all IPv4 records to be processed as IPv6 records using IPv4-mapped addresses and expands the formatting space to that required by IPv6 addresses.

The address policy can be specified in two ways: via the `SILK_IPV6_POLICY` environment variable or the `--ipv6-policy` parameter for the relevant tools. In general, the environment variable method tends to be preferred because it avoids further extending the length of SiLK commands.

Example 3.40: Changing Record Display with `--ipv6-policy`

```
<1>$ rwcut sample-v4+v6.rw --fields=1-5 --ipv6-policy=mix
                     sIP|                     dIP|sPort|dPort|pro|
            198.51.100.2|             203.0.113.3|   80|60014|  6|
      ::ffff:203.0.113.3|     ::ffff:198.51.100.2|60014|   80|  6|
         2001:db8::10:10|         2001:db8::10:12|   25|62123|  6|
         2001:db8::10:12|         2001:db8::10:10|62123|   25|  6|
     ::ffff:198.51.100.2|         2001:db8::10:10|    0|  256| 58|
<2>$ rwcut sample-v4+v6.rw --fields=1-5 --ipv6-policy=ignore
            sIP|            dIP|sPort|dPort|pro|
   198.51.100.2|    203.0.113.3|   80|60014|  6|
<3>$ rwcut sample-v4+v6.rw --fields=1-5 --ipv6-policy=only
                     sIP|                     dIP|sPort|dPort|pro|
      ::ffff:203.0.113.3|     ::ffff:198.51.100.2|60014|   80|  6|
         2001:db8::10:10|         2001:db8::10:12|   25|62123|  6|
         2001:db8::10:12|         2001:db8::10:10|62123|   25|  6|
     ::ffff:198.51.100.2|         2001:db8::10:10|    0|  256| 58|
<4>$ rwcut sample-v4+v6.rw --fields=1-5 --ipv6-policy=asv4
            sIP|            dIP|sPort|dPort|pro|
   198.51.100.2|    203.0.113.3|   80|60014|  6|
    203.0.113.3|   198.51.100.2|60014|   80|  6|
<5>$ rwcut sample-v4+v6.rw --fields=1-5 --ipv6-policy=force
                     sIP|                     dIP|sPort|dPort|pro|
     ::ffff:198.51.100.2|      ::ffff:203.0.113.3|   80|60014|  6|
      ::ffff:203.0.113.3|     ::ffff:198.51.100.2|60014|   80|  6|
         2001:db8::10:10|         2001:db8::10:12|   25|62123|  6|
         2001:db8::10:12|         2001:db8::10:10|62123|   25|  6|
     ::ffff:198.51.100.2|         2001:db8::10:10|    0|  256| 58|
```

Chapter 4

Using the Larger SiLK Tool Suite

The previous chapter described the basic SiLK tools and how to use them; with the knowledge from that chapter and a scripting language (e.g., Bash or Python®), an analyst can perform many forms of traffic analysis using flow records. However, both to speed up and simplify analyses, the SiLK suite includes a variety of additional analytical tools. This chapter describes the other tools in the analysis suite and explains how to use them. Like the previous chapter, this one will introduce these tools, present a series of example analyses, and briefly summarize the functions of the common parameters specific to each tool. Section 4.9 surveys parameters that are common across several tools, with further information presented in Section 3.9.

On completion of this chapter, you will be able to

- track flow record files with `rwfileinfo` and manipulate them with `rwcat`, `rwappend`, `rwdedupe`, and `rwsplit`
- create flow record files from text data with `rwtuc`
- translate packet data to flow records with `rwptoflow` and use the flows to isolate packets of interest with `rwpmatch`
- manipulate blocks of IP addresses in the SiLK suite via `rwnetmask`
- produce address-based representations for analysis using the set and bag tools
- associate related flow records via `rwmatch` and `rwgroup`
- annotate IP addresses with analysis information using prefix maps
- extend the function of SiLK tools by invoking plug-ins

4.1 Manipulating Flow Record Files

Once data are pulled from the repository, analysis may require these data to be extracted, rearranged, and combined with other flow data. This section describes the group of SiLK tools that manipulate flow record files: `rwcat`, `rwappend`, `rwsplit`, `rwdedupe`, `rwfileinfo`, and `rwtuc`.

4.1.1 Combining Flow Record Files with `rwcat` and `rwappend`

Example 4.1 profiles flow records for traffic with large aggregate volumes by the duration of transfer and by protocol. Even though subdividing files by repeated `rwfilter` calls allows the analyst to drill down to specific behavior, combining flow record files aids in providing context. The SiLK tool suite provides two tools specifically for combining flow record files:[14] `rwcat`, which concatenates flow record files in the order in which they are provided (see Figure 4.1), and `rwappend`, which places the contents of the flow record files at the end of the first specified flow record file (see Figure 4.2).

Since `rwcat` creates a new file, it can record annotation (using `--note-add` and `--note-file-add`) in the output file header. However, as discussed in the section for `rwfileinfo`, `rwcat` does not preserve this information from its input files. Since `rwappend` normally does not create a new file, it cannot add annotations and command history to the output file.

Figure 4.1: Summary of `rwcat`

rwcat

Description	Concatenates SiLK flow record files and copies to a new file
Call	`rwcat someflows.rw moreflows.rw --output-path=allflows.rw`
Parameters	`--ipv4-output` Converts IPv6 records to IPv4 records following the `asv4` policy in Figure 3.17 on page 76; ignores other IPv6 records `--byte-order` Writes the output in this byte order. Possible choices are `native` (the default), `little`, and `big` For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

Figure 4.2: Summary of `rwappend`

rwappend

Description	Appends the flow records from the successive files to the first file
Call	`rwappend allflows.rw laterflows.rw`
Parameters	`--create` Creates the output file if it does not already exist. Determines the format and version of the output file from the flow record file optionally named in this parameter `--print-statistics` Prints to standard error the count of records that are read from each input file and written to the output file For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

In Example 4.1, `rwcat` is used to combine previously filtered flow record files to permit the counting of overall values. In this example, the calls to `rwfilter` in Command 1 pull out all records. These records are then split into three files, depending on the duration of the flow: slow (at least 20 minutes), medium (1–20

[14] Many of the tools in the suite accept multiple flow record files as input to a single call. As examples, `rwfilter` will accept several flow files to filter in a single call; `rwsort` will accept several flow files to merge and sort in a single call. Very often, it is more convenient to use multiple inputs than to combine flow files.

minutes), and fast (less than 1 minute). 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). The calls to `rwcat` in Commands 5–7 combine the three splits for each protocol into one overall file per protocol. The remaining command produces a summary output reflecting record volumes.

Example 4.1: `rwcat` for Combining Flow Record Files

```
<1>$ rwfilter --type=in,inweb --start-date=2009/4/20T12 --end-date=2009/4/20T13 \
              --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
   echo -n "protocol, all, fast, med, slow"
   sed -e "s/^ *count-records - */$p, /" \
       -e "s/\([^,]\)  */\1, /g" ,c.txt
   done

Protocol, all, fast, med, slow
6, 783, 772, 11, 0
17, 157, 149, 0, 8
1, 7, 3, 4, 0
```

4.1.2 Merging While Removing Duplicate Flow Records with `rwdedupe`

When merging files that come from different sensors, occasionally analysts need to deal with having the same flow record collected by separate sensors. While this multiple recording is sometimes useful for traceability, more often it will distort the results of analysis. Duplicate records, whether caused by flows passing through multiple sensors or some other reason, can be removed efficiently with the `rwdedupe` command. Records

having identical address, port, and protocol information and close timings and sizes are considered duplicates. Exactly what "close" means is specified using the syntax and common parameters shown in Figure 4.3.

Figure 4.3: Summary of `rwdedupe`

rwdedupe

Description	Removes duplicate flow records
Call	`rwdedupe S1.rw S2.rw --sTime-delta=100 --ignore-fields=sensor --output-path=S1+2.rw`
Parameters	`--ignore-fields` Ignores these fields when comparing records. Values may be 1–10, 12–15, 20–21, 26–29, or their equivalent names (see Table 3.11 on page 59). `--packets-delta` Treats the `packets` field as identical if the values differ by this number of packets or less `--bytes-delta` Treats the `bytes` field as identical if the values differ by this number of bytes or less `--stime-delta` Treats the `sTime` field as identical if the values differ by this number of milliseconds or less `--duration-delta` Treats the `duration` field as identical if the values differ by this number of milliseconds or less For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

Example 4.2 shows an example of using `rwdedupe`. In this example, Command 1 creates named pipes for efficient passing of records. Commands 2 and 3 retrieve records from `SITE1` and `SITE2` sensors, passing these records via the named pipes. Command 4 merges the two groups of records and does protocol counts before and after applying `rwdedupe`. Command 5 pauses while the filtering and counting completes. Commands 6 and 7 show the results of the protocol counts with the small difference between results due to excluding duplicate records.

4.1.3 Dividing Flow Record Files with `rwsplit`

In addition to being able to join flow record files, some analyses are facilitated by dividing or sampling flow record files. To facilitate coarse parallelism, one approach 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. Figure 4.4 provides an overview of the syntax of `rwsplit` and a summary of its most common parameters. On each call, the `--basename` is required, and one of these parameters must be present: `--ip-limit`, `--flow-limit`, `--packet-limit`, or `--byte-limit`.

Example 4.3 is an example of a coarsely parallelized process. Command 1 pulls a large number of flow records and then divides those records into a series of 10,000-record files. In Command 3, each of these files is then fed in parallel to an `rwfilter` call to separate the flows based on the type of external site contacted. Applying these in parallel decreases the execution time of the analysis. To support the profiling in Command 5, a list of the generated filenames is initialized in Command 2 (to a marker value that will be ignored later) and then created during Command 3, using a Bash array structure. Command 4 waits for the

Example 4.2: `rwdedupe` for Removing Duplicate Records

```
<1>$ mkfifo /tmp/dedupe1.fifo /tmp/dedupe2.fifo
<2>$ rwfilter --sensor=SITE1 --type=in,inweb \
              --start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0- \
              --pass=/tmp/dedupe1.fifo &
<3>$ rwfilter --sensor=SITE2 --type=in,inweb \
              --start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0- \
              --pass=/tmp/dedupe2.fifo &
<4>$ rwcat /tmp/dedupe1.fifo /tmp/dedupe2.fifo --ipv4-output \
      | rwuniq --fields=protocol --values=records --output-path=dupe-1+2.txt \
               --copy-input=stdout \
      | rwdedupe --stime-delta=500 --ignore-fields=sensor \
      | rwuniq --fields=protocol --values=records --output-path=nodupe-1+2.txt &
<5>$ wait
<6>$ cat dupe-1+2.txt
pro|   Records|
  6|       783|
  1|         7|
 17|       157|
<7>$ cat nodupe-1+2.txt
pro|   Records|
  6|       685|
  1|         7|
 17|       157|
```

Figure 4.4: Summary of `rwsplit`

rwsplit

Description	Divides the flow records into successive files
Call	`rwsplit allflows.rw --flow-limit=1000 --basename=sample`
Parameters	Choose one: **`--ip-limit`** Specifies the IP address count at which to begin a new sample file **`--flow-limit`** Specifies the flow count at which to begin a new sample file **`--packet-limit`** Specifies the packet count at which to begin a new sample file **`--byte-limit`** Specifies the byte count at which to begin a new sample file Options: **`--basename`** Specifies the base name for output sample files (required) **`--sample-ratio`** Specifies the denominator for the ratio of records read to number written in a sample file (e.g., 100 means to write 1 out of 100 records) **`--seed`** Seeds the random number generator with this value **`--file-ratio`** Specifies the denominator for the ratio of sample filenames generated to the total number written (e.g., 10 means 1 of every 10 files will be saved) **`--max-outputs`** Specifies the maximum number of files to write to disk For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

parallel executions to complete. Command 5 profiles the split files by segment of the original file, generating a comma-separated table.

Example 4.4 is an example of a sampled-flow process. These commands estimate the percentage of UDP traffic moving across a large infrastructure over a workday. Command 1 does the initial data pull, retrieving a very large number of flow records, and then pulls 100 samples of 1,000 flow records each, with a 1% rate of sample generation (that is, of 100 samples of 1,000 records, only one sample is retained). Command 3 then summarizes each sample to isolate the percentage of UDP traffic in the sample, and the resulting percentages are profiled in Commands 5 through 7 to report the minimum, maximum, and median percentages.

4.1.4 Keeping Track of File Characteristics with `rwfileinfo`

Analyses using the SiLK tool suite can become quite complex, with several intermediate products created while isolating the behavior of interest. One tool that can aid in managing these products is `rwfileinfo`, which displays a variety of characteristics for each file format produced by the SiLK tool suite. See Figure 4.5 for parameter information. Some of these characteristics are shown in Example 4.5. `rwfileinfo` has a `--fields` parameter to allow analysts to specify the characteristics they want to see, as shown in Command 2 of Example 4.5. For most analysts, the three most important characteristics are the record count, file size, and command-line information. 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 `count-records` 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 IP set file. The `file-size` field displays the size of the file on disk. The `command-lines` field shows the commands used to generate the file.

Flow record files produced by `rwfilter` maintain an historical record that can be used to trace how and where a file was created. This information can be extracted using the `rwfileinfo` command. Example 4.5 shows an example of the results from several `rwfileinfo` commands. This field consists of a list of commands in historical order. This list includes `rwfilter` and a variety of other tools that add and preserve the command history. One tool that does not preserve this history (in case it needs to be cleared for any reason) is `rwcat`. Example 4.6 shows how the `command-lines` field expands with progressive commands and how `rwcat` does not preserve this history information.

An `annotations` characteristic is supported by the `--note-add` parameter of many tools in SiLK, as shown in Command 6 of Example 4.6. Annotations can be displayed using `rwfileinfo`.

While `rwfileinfo` is often associated with flow record files, it can also show information on sets, bags, and pmaps (see Section 4.4, Section 4.5, and Section 4.7, respectively). Example 4.7 displays how these files are handled. In Commands 1 and 2, a set and a bag for the example are created. Command 3 shows a full `rwfileinfo` result for the set file. Commands 5 through 7 show just the specific information for the set, bag, and pmap file, respectively. The pmap file was generated outside of these commands with an internal name `example-pmap`. If a prefix map was created without a mapname, `rwfileinfo` will return an empty result for the prefix-map-specific field.

Example 4.3: `rwsplit` for Coarse Parallel Execution

```
<1>$ rwfilter --type=inweb,outweb --start-date=2009/4/1 \
              --end-date=2009/4/30 --bytes-per-packet=45-  --pass=stdout \
      | rwsplit --flow-limit=10000 --basename=part
<2>$ s_list=(skip) # keep track of files generated
<3>$ 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[*]} $t{-miss,-threat,-casual}.rw)
  done
<4>$ wait
<5>$ echo "Part-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, 531, 5205, 4264,
part.00000001, 299, 5076, 4625,
part.00000002, 1004, 3157, 5839,
part.00000003, 141, 2146, 7713,
part.00000004, 60, 9436, 504,
part.00000005, 48, 8798, 1154,
part.00000006, 174, 8041, 1785,
part.00000007, 122, 8949, 929,
part.00000008, 258, 4507, 5235,
part.00000009, 197, 8677, 1126,
part.00000010, 248, 4092, 2087,
```

Example 4.4: `rwsplit` to Generate Statistics on Flow Record Files

```
<1>$ rwfilter --type=in,inweb --start-date=2009/4/1 \
              --end-date=2009/4/30 --protocol=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*; 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%:   6.400000
<6>$ echo -n "Min UDP%: " ; \
     tail -n 1 tmp.txt
Min UDP%:   0.300000
<7>$ echo -n "Median UDP%: "; \
     head -n 50 tmp.txt \
      | tail -n 1
Median UDP%:   0.300000
```

Example 4.5: `rwfileinfo` for Display of Flow Record File Characteristics

```
<1>$ rwfileinfo medfile.rw
medfile.rw:
  format(id)          FT_RWIPV6ROUTING(0x0c)
  version             16
  byte-order          littleEndian
  compression(id)     none(0)
  header-length       352
  record-length       88
  record-version      1
  silk-version        3.5.1
  count-records       15
  file-size           1672
  command-lines
                  1  rwfilter --sensor=S0 --type=in,inweb
--start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0-
--note-add=example --pass=stdout
                  2  rwfilter --duration=1200- --pass=slowfile.rw
--fail=stdout stdin
                  3  rwfilter --duration=60-1199 --pass=medfile.rw
--fail=fastfile.rw stdin
  annotations
                  1  example
<2>$ rwfileinfo medfile.rw --fields=count-records
medfile.rw:
  count-records       15
```

Figure 4.5: Summary of `rwfileinfo`

rwfileinfo

Description Displays summary information about one or more SiLK files

Call `rwfileinfo allflows.rw --fields=count-records,command-lines`

Parameters **`--fields`** Selects which summary information to display via number or name (by default, all the available fields). Possible values include

#	Field	Description
1	`format`	Binary file format indicator
2	`version`	Version of file header
3	`byte-order`	Byte order of words written to disk
4	`compression`	Type of space compression used
5	`header-length`	Number of bytes in file header
6	`record-length`	Number of bytes in fixed-length records
7	`count-records`	Number of records in the file unless record-length=1
8	`file-size`	Total number of bytes in the file on disk
9	`command-lines`	List of stored commands that generated this file
10	`record-version`	Version of records in file
11	`silk-version`	Software version of SiLK tool that produced this file
12	`packed-file-info`	Information from packing process
13	`probe-name`	Probe info for files created by flowcap
14	`annotations`	List of notes
15	`prefix-map`	Prefix map name and header version
16	`ipset`	IP-set format information
17	`bag`	Bag key and count information

`--summary` Prints a summary of total files, file sizes, and records

For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

Example 4.6: `rwfileinfo` for Showing Command History

```
<1>$ rwfileinfo slowfile.rw --field=command-lines
slowfile.rw:
  command-lines
                    1  rwfilter --sensor=SEN --type=in,inweb
--start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0-
--note-add=example --pass=stdout
                    2  rwfilter --duration=1200- --pass=slowfile.rw
--fail=stdout stdin
<2>$ rwfilter slowfile.rw --protocol=6 --dport=22 --pass=newfile.rw
<3>$ rwfileinfo newfile.rw --field=command-lines
newfile.rw:
  command-lines
                    1  rwfilter --sensor=SEN --type=in,inweb
--start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0-
--note-add=example --pass=stdout
                    2  rwfilter --duration=1200- --pass=slowfile.rw
--fail=stdout stdin
                    3  rwfilter --protocol=6 --dport=22 --pass=newfile.rw
slowfile.rw
<4>$ rwsort newfile.rw --fields=sTime --output-path=sorted.rw
<5>$ rwfileinfo sorted.rw --field=command-lines
sorted.rw:
  command-lines
                    1  rwfilter --sensor=SEN --type=in,inweb
--start-date=2009/4/20T12 --end-date=2009/4/20T13 --protocol=0-
--note-add=example --pass=stdout
                    2  rwfilter --duration=1200- --pass=slowfile.rw
--fail=stdout stdin
                    3  rwfilter --protocol=6 --dport=22 --pass=newfile.rw
slowfile.rw
                    4  rwsort --fields=sTime --output-path=sorted.rw
newfile.rw
<6>$ rwcat sorted.rw --output-path=new2.rw \
          --note-add="originally from slowfile.rw, filtered for dport 22/TCP
<7>$ rwfileinfo new2.rw --field=command-lines,annotations
new2.rw:
  annotations
                    1  originally from slowfile.rw, filtered for dport
22/TCP
```

Example 4.7: `rwfileinfo` for Sets, Bags, and Prefix Maps

```
<1>$ rwfilter --type=out,outweb --start-date=2009/4/21 \
              --protocol=0- --pass=stdout \
      | rwset --sip-file=internal.set
<2>$ rwfilter --type=out,outweb --start=2009/4/21 \
              --protocol=0- --pass=stdout \
      | rwbag --sip-flow=internal.bag
<3>$ rwfileinfo internal.set
internal.set:
  format(id)          FT_IPSET(0x1d)
  version             16
  byte-order          littleEndian
  compression(id)     none(0)
  header-length       175
  record-length       1
  record-version      3
  silk-version        3.5.1
  count-records       3328
  file-size           3503
  ipset               16-way branch, root@3, 33 x 80b nodes, 86 x 8b leaves
  command-lines
                  1  rwfilter --type=out,outweb --start=2009/4/21
--protocol=0- --pass=stdout
                  2  rwset --sip-file=internal.set
<4>$ rwfileinfo internal.set --fields=ipset
internal.set:
  ipset               16-way branch, root@3, 33 x 80b nodes, 86 x 8b leaves
<5>$ rwfileinfo internal.bag --fields=bag
internal.bag:
  bag                 key: sIPv4 @ 4 octets; counter: records @ 8 octets
<6>$ rwfileinfo internal.pmap --fields=prefix-map
internal.pmap:
  prefix-map          v1: example-pmap
```

4.1.5 Creating Flow Record Files from Text with `rwtuc`

The `rwtuc` (Text Utility Converter) tool allows creating SiLK flow record files from columnar text information. `rwtuc`, effectively, is the inverse of `rwcut`, with additional parameters to supply values not given by the columnar input. Figure 4.6 provides an overview of the syntax for `rwtuc`.

Figure 4.6: Summary of `rwtuc`

rwtuc

Description	Generates SiLK flow records from textual input similar to `rwcut` output
Call	`rwtuc flows.rw.txt --output-path=flows.rw`
Parameters	**`--fields`** Specifies the fields to parse from the input. Values may be • a field number: 1–15, 20–21, 26–29 • a field name equivalent to one of the field numbers above (see Table 3.11 on page 59) • the keyword **`ignore`** for an input column not to be included in the output records. The parameter is unnecessary if the input file has appropriate column headings. **`--bad-input-lines`** Specifies the file or stream to write each bad input line to (filename and line number prepended) **`--verbose`** Prints an error message for each bad input line to standard error **`--stop-on-error`** Prints an error message for a bad input line to standard error and exits **`--`*`fixed-value-parameter`*** Uses the value as a fixed value for this field in all records. See Table 4.1 for the parameter name for each field. For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

`rwtuc` is useful in several scenarios. Some scripting languages (Perl in particular) have string-processing functions that may be used for analysis, but for compactness and speed of later processing a binary result may be needed. Therefore, `rwcut` would be used to convert the binary flow record files to text, the scripting language would process it, and `rwtuc` would convert the text output back to the binary flow record format. However, if the scripting can be done in the Python programming language, the programming interface contained in the `silk` module allows direct manipulation of the binary structures without the aforementioned conversion to text or the following conversion to binary. This binary manipulation is more efficient than a text-based form.[15]

Alternatively, if exchanging data requires cleansing a flow record file, `rwtuc` provides the analyst complete control of the binary representation's content.[16] By converting to text, performing any required edits on it, and then generating a binary representation from the edited text, an analyst can ensure that no unreleasable content is present in the binary form. Example 4.8 shows a sample use of `rwtuc` for cleansing the binary

[15] There are several published examples where individuals encoded non-flow information as binary flow records using `rwtuc` or PySiLK so that SiLK commands can be used for the fast filtering and processing of that information.

[16] 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 help to cleanse IP sets.

Table 4.1: Fixed-Value Parameters for `rwtuc`

Field	Parameter Name	Field	Parameter Name
sIP	--saddress	in	--input-index
dIP	--daddress	out	--output-index
sPort	--sport	nhIP	--next-hop-ip
dPort	--dport	class	--class
protocol	--protocol	type	--type
packets	--packets	iType	--icmp-type
bytes	--bytes	iCode	--icmp-code
flags	--flags-all	initialFlags	--flags-initial
sTime	--stime	sessionFlags	--flags-session
duration	--duration	attributes	--attributes
eTime	--etime	application	--application
sensor	--sensor		

representation. After `rwtuc` is invoked in Command 3, both the file-header information and non-preserved fields have generic or null values.

`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 4.8, 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.

Example 4.8: `rwtuc` for Simple File Cleansing

```
<1>$ rwfilter --sensor=SEN1 --type=in --start-date=2009/4/20 \
             --protocol=17 --bytes-per-packet=100-  --pass=bigflows.rw
<2>$ rwcut bigflows.rw --fields=1-10 --num-recs=20 \
     | sed -re "s/([0-9]+\.){3}/10.3.2./g" >bigflows.rw.txt
<3>$ rwtuc bigflows.rw.txt --output-path=cleansed.rw
<4>$ rwfileinfo cleansed.rw
cleansed.rw:
  format(id)          FT_RWIPV6ROUTING(0x0c)
  version             16
  byte-order          littleEndian
  compression(id)     none(0)
  header-length       88
  record-length       88
  record-version      1
  silk-version        3.5.1
  count-records       20
  file-size           1848
  command-lines
                  1  rwtuc --output-path=cleansed.rw bigflows.rw.txt
<5>$ rwcut cleansed.rw --fields=sIP,dIP,sTime,sensor --num-recs=4
            sIP|            dIP|                  sTime|sen|
    10.3.2.176|     10.3.2.131|2009/04/20T11:42:10.268|   |
    10.3.2.176|     10.3.2.131|2009/04/20T12:11:15.996|   |
    10.3.2.176|     10.3.2.131|2009/04/20T12:12:31.511|   |
    10.3.2.176|     10.3.2.131|2009/04/20T12:14:25.029|   |
```

4.2 Analyzing Packet Data with rwptoflow and rwpmatch

Analysts use the `rwptoflow` and `rwpmatch` tools to generate single-packet flow records from packet content (i.e., PCAP) data, analyze and filter those flow records using the SiLK tools, and subsequently filter the packet data based on that analysis. Third-party tools, such as `ngrep` (http://ngrep.sourceforge.net) may also filter packet content data based on regular expressions.

Another option for processing packets is to aggregate the packets into true flow records. Analysts can do that through the `rwp2yaf2silk` tool that uses the features of `rwtuc` and the `yaf` and `yafscii` tools (the latter two are available from http://tools.netsa.cert.org/yaf/). Once converted to flow records, all the SiLK tools can process them as if they were from the repository, but it is difficult to re-identify packets with processed flow records. For analyses that involve both packet and flow analysis, `rwptoflow` and `rwpmatch` are preferred.

4.2.1 Creating Flows from Packets Using rwptoflow

The `rwptoflow` tool generates a single-packet flow record for every IP packet in a packet capture file. The packet formats do not contain routing information, which is available in some flow record formats. The values for routing-information flow record fields may be set for the generated flows using the parameters `--set-sensorid`, `--set-inputindex`, `--set-outputindex`, and `--set-nexthopip`. For example, it is possible to set the sensor ID manually for a packet content source so that network flow data combined from several sensors can be filtered or sorted by the sensor value later. `rwptoflow` is summarized in Figure 4.7. `rwptoflow` with `--active-time` can be used to specify the generation of flows only for a specific time interval of interest. During this time interval, `--packet-pass-output` and `--packet-reject-output` can be used to produce packet files that were either converted to flows or not converted to flows. Finally, the `--plugin` parameter can be used to incorporate plug-ins for additional functionality in packet conversion, analogous to `rwfilter` plug-ins.[17]

A packet might not be converted to a flow record for these reasons:

- The packet is not for an Internet protocol (IP or its encapsulated protocols). LAN-based protocols (such as the Address Resolution Protocol (ARP) are not Internet protocols. As such, there isn't enough information in the packet to build a flow record for it. Other tools, such as `tcpdump` or Wireshark® can be used to examine and analyze these packets.
- The packet is erroneous, and the information used to build a flow record is inconsistent in a way that prevents record generation. This may happen because of transmission problems with the packet or because the capture file may have been corrupted.
- The packet capture snapshot length isn't large enough to capture all the needed fields. If a very short snapshot length is used, not all of the header may be captured. Therefore, the captured packet may not contain enough information to build a flow record for it.
- The IP packets are encapsulated in IEEE 802.1Q Virtual LAN (VLAN) frames.

Any of these will cause the packet to be rejected. Example 4.9 shows a simple conversion of the `packets.pcap` capture file into the `mypkts.rw1` flow record file, restricting the conversion to a specific time period and producing dumps of packets converted (`mypkts.dmp`) and rejected (`mypkts-bad.dmp`).

[17] In the current version, PySiLK plug-ins are not implemented for `rwptoflow`. All plug-ins currently must use the C language API.

Figure 4.7: Summary of `rwptoflow`

rwptoflow

Description	Reads a packet capture file and generates a SiLK flow record for every packet
Call	`rwptoflow packets.pcap --flow-output=single_pkt_flows.rw1`
Parameters	**`--active-time`** Sets the time interval of interest **`--packet-pass-output`** Specifies a path for valid packets in the time interval of interest **`--packet-reject-output`** Specifies a path for invalid packets in the time interval of interest **`--plugin`** Specifies a plug-in to be used in the conversion **`--flow-output`** Writes the generated SiLK flow records to the specified path or `stdout` if the path is not given **`--reject-all-fragments`** Does not generate a SiLK flow record for any fragmented packets **`--reject-nonzero-fragments`** Does not generate a SiLK flow record for any packet with a non-zero fragment offset **`--reject-incomplete`** Does not generate a SiLK flow record for any zero-fragment or unfragmented packets when the record cannot be completely filled (missing ICMP type & code, TCP/UDP ports, TCP flags) **`--set-sensorid`** Sets the sensor ID for all flows (0–65,534). When not specified, the ID is set to 65,535. **`--set-inputindex`** Sets the input SNMP index for all flows, 0–65,535. **`--set-outputindex`** Sets the output SNMP index for all flows, 0–65,535. **`--set-nexthopip`** Sets the next-hop IP address for all flows. **`--print-statistics`** Prints the count of packets read, packets processed, and bad packets to standard error For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

Example 4.9: `rwptoflow` for Simple Packet Conversion

```
<1>$ rwptoflow packets.pcap --active-time=2013/8/25T05:27-2013/8/25T05:45 \
      --packet-pass-output=mypkts.dmp --packet-reject-output=mypkts-bad.dmp \
      --flow-output=mypkts.rw1
```

4.2.2 Matching Flow Records with Packet Data Using `rwpmatch`

`rwpmatch` takes a packet capture input file and filters it based on flow records from a SiLK flow record file. It is designed to allow flow records from `rwptoflow` (that are filtered or processed) to be matched with the packet content data that produced them. The resulting packet capture file is output on standard output.

The flow record file input to `rwpmatch` should contain single-packet flow records (e.g., those originally derived from a packet capture file using `rwptoflow`). If a flow record is found that does not represent a corresponding packet record, `rwpmatch` will return an error. Both the packet capture and the flow record file inputs must be time-ordered. The syntax of `rwpmatch` is summarized in Figure 4.8. By default, `rwpmatch` will consider only the source address, destination address, and the time to the second. By using the `--ports-compare` parameter, the source and destination port can also be considered in the match. By using the `--msec-compare` parameter, time will be compared to the millisecond.

Figure 4.8: Summary of `rwpmatch`

rwpmatch

Description	Matches a packet capture file against a SiLK flow record file that has a flow for every packet, producing a new packet capture file on standard output
Call	`rwpmatch packets.pcap --flow-file=selected.rw1 >selected.pcap`
Parameters	**`--flow-file`** Specifies the flow record file to be used in the match (required) **`--ports-compare`** Uses port information in the match **`--msec-compare`** Uses milliseconds in the match For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

`rwpmatch` is I/O intensive. The tool works by reading an entire packet capture file and the entire flow record file. It may be worthwhile to optimize an analysis process to avoid using `rwpmatch` until payload filtering is necessary. Saving the output from `rwpmatch` as a partial-results file and comparing that file to files generated by later steps in the analysis (rather than comparing the later results against the original packet capture file) can also provide significant performance gains.

The packet-analysis tools are typically used in combination with payload-filtering tools, like `ngrep`, that allow an analyst to partition traffic based on payload signatures prior to using the SiLK tools for analysis or, conversely, to identify a traffic phenomenon (e.g., worm propagation) through flow analysis and then filter the packets that correspond to the flow records that make up that traffic.

In Example 4.10, the `data.pcap` packet capture file is filtered by the `sip.set` IP-set file by converting it to a SiLK flow record file, filtering the flows by the source IP addresses found in the set, and then matching the original packet capture file against the filtered flow file.

Example 4.10: `rwptoflow` and `rwpmatch` for Filtering Packets Using an IP Set

```
<1>$ rwptoflow data.pcap --flow-output=data.rw1
<2>$ rwfilter data.rw1 --sipset=sip.set --pass=filtered.rw1
<3>$ rwpmatch data.pcap --flow-file=filtered.rw1 >filtered.pcap
```

4.3 Aggregating IP Addresses by Masking with `rwnetmask`

When working with IP addresses and utilities such as `rwuniq` and `rwstats`, an analyst will sometimes want to analyze activity across *networks* rather than individual IP addresses (for example, all the activity originating from the /24s constituting the enterprise network rather than generating an individual entry for each address). To do so, SiLK provides a tool called `rwnetmask` that can reduce IP addresses to prefix values of a parametrized length. Figure 4.9 summarizes the syntax of `rwnetmask`.

Figure 4.9: Summary of `rwnetmask`

rwnetmask

Description	Zeroes all bits past the specified prefix length on the specified address in SiLK flow records
Call	`rwnetmask flows.rw --4sip-prefix-length=24 --output-path=sip-24.rw`
Parameters	Choose one or more: **`--4sip-prefix-length`** Gives number of high bits of source IPv4 address to keep **`--4dip-prefix-length`** Gives number of high bits of destination IPv4 to keep **`--4nhip-prefix-length`** Gives number of high bits of next-hop IPv4 to keep **`--6sip-prefix-length`** Gives number of high bits of source IPv6 to keep **`--6dip-prefix-length`** Gives number of high bits of destination IPv6 to keep **`--6nhip-prefix-length`** Gives number of high bits of next-hop IPv6 to keep For additional parameters, see Table 4.4 on page 134 and Table 3.14 on page 72.

The query in Example 4.11 is followed by an `rwnetmask` call to retain only the first 16 bits (two octets) of source IPv4 addresses, as shown by the `rwcut` output.

Example 4.11: `rwnetmask` for Abstracting Source IPv4 addresses

```
<1>$ rwfilter --type=inweb --start-date=2009/4/20T12 --end-date=2009/4/20T13 \
               --protocol=6 --dport=80 --max-pass-records=3 --pass=stdout \
      | rwnetmask --4sip-prefix-length=16 \
      | rwcut --fields=1-5
             sIP|            dIP|sPort|dPort|pro|
     198.51.0.0|192.168.244.120|57675|   80|  6|
     198.51.0.0|192.168.244.120|62922|   80|  6|
     198.51.0.0|192.168.244.120|54830|   80|  6|
```

As Example 4.11 shows, `rwnetmask` replaces the last 16 bits[18] of the source IP address with zero, so all IP addresses in the 198.51/16 block (for example) will be masked to produce the same IP address. Using `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.

[18] 32 bits total for an IPv4 address minus the 16 bits specified in the command for the prefix length leaves 16 bits to be masked.

4.4 Summarizing Traffic with IP Sets

Up to this point, this handbook has focused exclusively on raw SiLK flow records: traffic that can be accessed and manipulated using the SiLK tools. This section focuses on initial summary structures: IP sets.

The IP-set tools provide facilities for manipulating *summaries* of data and manipulate arbitrary collections of IP addresses. These sets can be generated from network flow data or via text files created by users or obtained from other sources.

4.4.1 What Are IP Sets?

An IP set is a data structure that represents an arbitrary collection of individual IP addresses.[19] For example, an IP set could consist of the addresses {1.1.1.3, 92.18.128.22, 125.66.11.44} or all the addresses in a single /24 netblock. There are numerous uses for these sets, including representing address spaces; identifying safe (whitelist) addresses or threatening (blacklist) addresses; or listing addresses related to a given activity. IP sets are binary representations of data. Using binary representations, sets can be manipulated efficiently and reliably. IP sets can be any mixture of IPv4 or IPv6 network addresses, although the `SILK_IPV6_POLICY` environment variable or `--ipv6-policy` values may affect processing of these addresses, as shown in Table 3.17 on page 76.

Because IP sets are binary objects, they are created and modified using special set tools: `rwset`, `rwsetbuild`, `rwsettool`, along with `rwbagtool --coverset` and some PySiLK scripts. They are read using `rwsetmember`, `rwsetcat`, `rwbagtool --intersect`, `rwbagtool --complement-intersect`, and `rwbagcat --mask-set`, as well as the set parameters in `rwfilter` (see Table 3.4 on page 34) and some PySiLK scripts. All these tools support both IPv4 and IPv6 addresses, although only `rwset` has an `--ipv6-policy` parameter.

4.4.2 Creating IP Sets with `rwset`

IP sets are created from flow records via `rwset`,[20] from text via `rwsetbuild`, or from bags via `rwbagtool`. (More information on `rwbagtool` is found in Section 4.5.5.) `rwset` is summarized in Figure 4.10.

`rwset` generates sets from filtered flow records. To invoke it, pipe output from `rwfilter` into `rwset`, as shown in Example 4.12.

Example 4.12: `rwset` for Generating an IP-Set File

```
<1>$ rwfilter medfile.rw --protocol=6 --pass=stdout \
      | rwset --dip-file=medtcp-dest.set
<2>$ file medtcp-dest.set
medtcp-dest.set: data
```

The call to `rwset` shown in Example 4.12 creates an IP-set file named `medtcp-dest.set` that consists of all the destination IP addresses for TCP records in `medfile.rw`. The `file` command shows that the result is a binary data file.

[19] IP sets are not only supported as part of the SiLK suite, but a separate distribution of the IP-set functionality is provided at http://tools.netsa.cert.org/silk-ipset/.

[20] IP sets can also be created from flow records using PySiLK or a deprecated tool called `rwaddrcount`.

Figure 4.10: Summary of `rwset`

rwset

Description	Generates IP-set files from flows
Call	`rwset flows.rw --sip-file=flows_sip.set`
Parameters	Choose one or more: **`--sip-file`** Specifies an IP-set file to generate with source IP addresses from the flows records **`--dip-file`** Specifies an IP-set file to generate with destination IP addresses from the flows records **`--nhip-file`** Specifies an IP-set file to generate with next-hop IP addresses from the flows records **`--any-file`** Specifies an IP-set file to generate with both source and destination IP addresses from the flows records Options: **`--record-version`** Specifies the version of records to write to a file. Allowed arguments are 0, 2, 3, and 4; record version affects backwards compatibility. It can also be set by the `SILK_IPSET_RECORD_VERSION` environment variable. **`--invocation-strip`** Does not copy command lines from the input files to the output files For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

An alternative method for generating sets is using the `rwsetbuild` tool. `rwsetbuild` reads a text file containing IP addresses and generates an IP-set file with those addresses (see Example 4.15 for sample calls).

4.4.3 Reading Sets with `rwsetcat`

The primary tool for reading sets is `rwsetcat`, which reads one or more set files and then displays the IP addresses in each file or prints out statistics about the set in each file. The basic invocation of `rwsetcat` is shown in Example 4.13. A summary of some common parameters is shown in Figure 4.11.

Example 4.13: `rwsetcat` to Display IP Sets

```
<1>$ rwsetcat medtcp-dest.set | head -n 5
192.168.45.27
192.168.61.26
```

Figure 4.11: Summary of `rwsetcat`

rwsetcat

Description	Lists contents of IP-set files as text on standard output
Call	`rwsetcat low_sip.set >low_sip.set.txt`
Parameters	**`--network-structure`** Prints the network structure of the set. Syntax: [`v6:`|`v4:`][*list-lengths*[`S`][`/`[*summary-lengths*]]] A length may be expressed as an integer prefix length (often preferred) or a letter: `T` for total address space (/0), `A` for /8, `B` for /16, `C` for /24, `X` for /27, and `H` for Host (/32 for IPv4, /128 for IPv6); with `S` for default summaries. **`--cidr-blocks`** Groups IP addresses that share a CIDR block **`--count-ips`** Prints the number of IP addresses. Disables default printing of addresses **`--print-statistics`** Prints set statistics (min-/max-IP address, etc.). Also disables default printing of addresses **`--print-ips`** Prints IP addresses when count or statistics parameter is given For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

In Example 4.13, the call to `rwsetcat` will print out all the addresses in the set; IP addresses appear in ascending order. In addition to printing out IP addresses, `rwsetcat` can also perform counting and statistical reporting, as shown in Example 4.14. These features are useful for describing the set without dumping out all 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`.

In Example 4.14, we also see the versatility of 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 small 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 smaller 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. In Command 14, IPv4 addresses are converted to IPv4-mapped IPv6 addresses, and other IPv6 addresses are allowed to be output. Command 15 shows that `v6:32` does not list individual hosts. Command 16 shows the default summary lengths for IPv6. In Command 17, the list length is larger than the smallest default summary length, as in Command 11. Command 18 shows how to display individual hosts in IPv6. Command 19 shows that for IPv6, `H` is equivalent to 128, because there are 128 bits in an IPv6 address.

4.4.4 Manipulating Sets with `rwsettool`, `rwsetbuild`, and `rwsetmember`

`rwsettool` is the primary tool used to manipulate constructed sets. It provides the most common set operations, working on arbitrary numbers of IP-set files. See Figure 4.12 for a summary of its syntax and most of its parameters. `rwsettool` will accept any number of set files as parameters, as long as there is at least one.

`rwsetbuild` creates IP-set files from text files. Example 4.15 creates two sets using `rwsetbuild`: one consisting of the IP addresses 1.1.1.1–5 and the other consisting of the IP addresses 1.1.1.3, 1.1.1.5, and 2.2.2.2.

`rwsettool --intersect` generates a set of addresses such that each address is present in *all* the IP-set files listed as arguments. Example 4.16 shows two sets intersected. Each set is specified by a filename as an argument. The resulting set is written to the `inter_result.set` file as shown in Command 1, with the results shown after Command 3. As the example shows, the resulting set consists of the IP addresses 1.1.1.3 and 1.1.1.5; the intersection of sets is the set of IP addresses present in every individual set.

In addition to straight intersection, `rwsettool` can also be used to subtract the contents of one set from another, using the `--difference` parameter as shown in Command 2 of Example 4.16. The resulting `sub_result.set` set is shown after Command 4. The `sub_result.set` file consists of all elements that were in `a.set` but *not* in `b.set`.

The IP-set union command is `rwsettool --union`. This takes any number of IP-set filenames as arguments and returns a set that consists of all IP addresses that appear in *any* of the files. Example 4.17, using `a.set` and `b.set`, demonstrates this capability.

`rwsetmember` allows easy testing for the presence of addresses matching a wildcard (of the sort acceptable to `rwfilter --saddress` and described in Section 3.2.1) in one or more IP-set files. Example 4.18 shows some examples of its use. `rwsetmember --count` prints counts of the number of matches, along with each filename. `rwsetmember --quiet` suppresses all output but sets the exit status to 0 if any of the files contains a matching address or to 1 if no such matches are found.

Example 4.14: `rwsetcat` Options for Showing Structure

```
<1>$ rwsetcat medtcp-dest.set --count-ips
2
<2>$ rwsetcat medtcp-dest.set --print-statistics
Network Summary
        minimumIP =   192.168.63.75
        maximumIP =  192.168.76.162
                2 hosts (/32s),    0.000000% of 2^32
                1 occupied /8,     0.390625% of 2^8
                1 occupied /16,    0.001526% of 2^16
                2 occupied /24s,   0.000012% of 2^24
                2 occupied /27s,   0.000001% of 2^27
<3>$ rwsetcat medtcp-dest.set --network-structure
TOTAL| 2 hosts in 1 /8, 1 /16, 2 /24s, and 2 /27s
<4>$ rwsetcat medtcp-dest.set --network-structure=4
      192.0.0.0/4| 2
<5>$ rwsetcat medtcp-dest.set --network-structure=18
    192.168.0.0/18| 1
   192.168.64.0/18| 1
<6>$ rwsetcat medtcp-dest.set --network-structure=4,18
  192.168.0.0/18    | 1
  192.168.64.0/18   | 1
192.0.0.0/4         | 2
<7>$ rwsetcat medtcp-dest.set --network-structure=0,18
  192.168.0.0/18    | 1
  192.168.64.0/18   | 1
TOTAL               | 2
<8>$ rwsetcat medtcp-dest.set --network-structure=4,18/24
  192.168.0.0/18    | 1 host in 1 /24
  192.168.64.0/18   | 1 host in 1 /24
192.0.0.0/4         | 2 hosts in 2 /18s and 2 /24s
<9>$ rwsetcat medtcp-dest.set --network-structure=4/
      192.0.0.0/4| 2 hosts
<10>$ rwsetcat medtcp-dest.set --network-structure=4S
      192.0.0.0/4| 2 hosts in 1 /8, 1 /16, 2 /24s, and 2 /27s
<11>$ rwsetcat medtcp-dest.set --network-structure=12S
    192.160.0.0/12| 2 hosts in 1 /16, 2 /24s, and 2 /27s
<12>$ rwsetcat medtcp-dest.set --network-structure=H
  192.168.63.75|
 192.168.76.162|
<13>$ rwsetcat medtcp-dest.set --network-structure=32
  192.168.63.75|
 192.168.76.162|
<14>$ rwsetcat medtcp-dest.set --network-structure=v6:
TOTAL| 2 hosts in 1 /48 and 1 /64
<15>$ rwsetcat medtcp-dest.set --network-structure=v6:32
                                    ::/32| 2
<16>$ rwsetcat medtcp-dest.set --network-structure=v6:32S
                                    ::/32| 2 hosts in 1 /48 and 1 /64
<17>$ rwsetcat medtcp-dest.set --network-structure=v6:56S
                                    ::/56| 2 hosts in 1 /64
<18>$ rwsetcat medtcp-dest.set --network-structure=v6:H
                   ::ffff:192.168.63.75|
                  ::ffff:192.168.76.162|
<19>$ rwsetcat medtcp-dest.set --network-structure=v6:128
                   ::ffff:192.168.63.75|
                  ::ffff:192.168.76.162|
```

Figure 4.12: Summary of `rwsettool`

rwsettool

Description	Manipulates IP-set files to produce new IP-set files
Call	`rwsettool --intersect src1.set src2.set --output-path=both.set`
Parameters	Choose one: **`--union`** Creates set containing IP addresses in any input file **`--intersect`** Creates set containing IP addresses in all input files **`--difference`** Creates set containing IP addresses from first file not in any of the remaining files **`--mask`** Creates set containing one IP address from each block of the specified bitmask length when any of the input IP sets have an IP address in that block **`--fill-blocks`** Creates an IP set containing a completely full block with the specified prefix length when any of the input IP sets have an IP address in that block **`--sample`** Creates an IP set containing a random sample of IP addresses from all input IP sets. Requires either the `--size` or `--ratio` option Options: **`--size`** Specifies the sample size (number of IP addresses sampled from each input IP set). Requires the `--sample` parameter **`--ratio`** Specifies the probability (as a floating point value between 0.0 and 1.0) that an IP address will be sampled. Requires the `--sample` parameter **`--seed`** Specifies the random number seed for the `--sample` parameter and is used only with that parameter **`--note-strip`** Does not copy notes from the input files to the output file **`--invocation-strip`** Does not copy command history from the input files to the output file **`--record-version`** Specifies SiLK Version 2 or Version 3 format for the records in the output file For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.15: `rwsetbuild` for Generating IP Sets

```
<1>$ echo "1.1.1.1-5" >a.set.txt
<2>$ cat <<END_FILE >b.set.txt
1.1.1.3
1.1.1.5
2.2.2.2
END_FILE
<3>$ rwsetbuild a.set.txt a.set
<4>$ rwsetbuild b.set.txt b.set
```

Example 4.16: `rwsettool` to Intersect and Difference IP Sets

```
<1>$ rwsettool --intersect a.set b.set --output-path=inter_result.set
<2>$ rwsettool --difference a.set b.set --output-path=sub_result.set
<3>$ rwsetcat inter_result.set
1.1.1.3
1.1.1.5
<4>$ rwsetcat sub_result.set
1.1.1.1
1.1.1.2
1.1.1.4
```

Example 4.17: `rwsettool` to Union IP Sets

```
<1>$ rwsettool --union a.set b.set --output-path=union_result.set
<2>$ rwsetcat union_result.set
1.1.1.1
1.1.1.2
1.1.1.3
1.1.1.4
1.1.1.5
2.2.2.2
```

Example 4.18: `rwsetmember` to Test for an Address

```
<1>$ rwsetmember 2.2.2.2 b.set
b.set
<2>$ rwsetmember 2.2.2.2 a.set
<3>$ rwsetmember 1.1.1.3 a.set b.set
a.set
b.set
<4>$ rwsetmember 1.1.1.3 a.set b.set --count
a.set:1
b.set:1
```

4.4.5 Using `rwsettool --intersect` to Fine Tune IP Sets

Using IP sets can focus on alternative representations of traffic and identify different classes of activity. Example 4.19 drills down on IP sets themselves and provides a different view of this traffic.

Example 4.19: Using `rwset` to Filter for a Set of Scanners

```
<1>$ rwfilter fastfile.rw --protocol=6 --packets=1-3 --pass=stdout \
      | rwset --sip-file=fast-low.set
<2>$ rwfilter fastfile.rw --protocol=6 --packets=4-  --pass=stdout \
      | rwset --sip-file=fast-high.set
<3>$ rwsettool --difference fast-low.set fast-high.set \
               --output-path=fast-only-low.set
<4>$ rwsetcat fast-low.set --count-ips
8
<5>$ rwsetcat fast-only-low.set --count-ips
5
```

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 `fast-only-low.set` set file consists of five IP addresses in contrast to the set of eight that produced low-packet flow records—these addresses are consequently suspicious.[21]

4.4.6 Using `rwsettool --union` to Examine IP-Set Growth

One way to use `rwsettool --union` is to track common addresses of hosts contacting a single site. A fictitious site collects IP address sets every hour. Example 4.20 replicates this collection by generating hourly IP sets for traffic reaching one server, using the Bash `for` statement in Command 1.

Running this loop results in 72 files, one for each hour. After this, Commands 2 and 3 in Example 4.20 use `rwsettool` to build the cumulative set of addresses across hours, which is iteratively counted using `rwsetcat`.

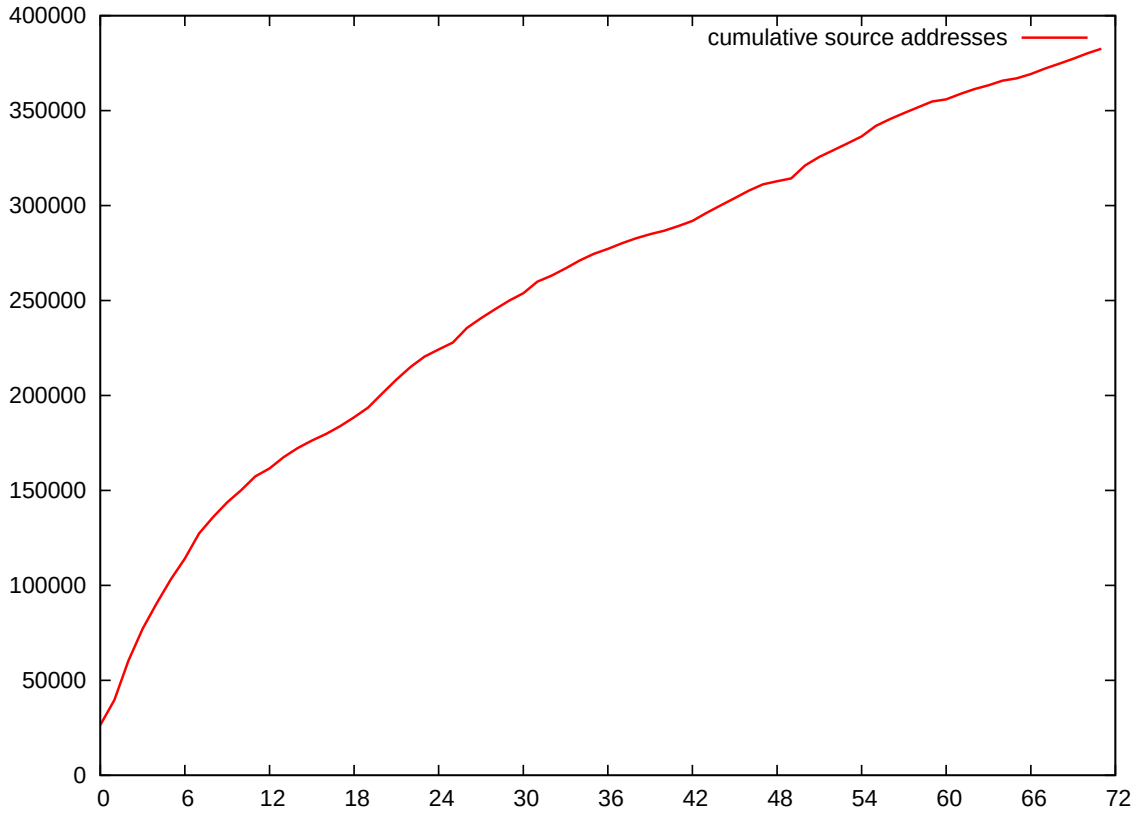
This loop starts by copying the first hour to initialize a temporary file (`buffer.set`). It then iterates through every set file in the directory, creating a union of that set file with the `buffer.set` file, and printing the total number of IP addresses from the union. The resulting file can then be plotted with `gnuplot`. The graph in Figure 4.13 shows the resulting image: the cumulative number of source IP addresses seen in each hour.

4.4.7 Backdoor Analysis with IP Sets

A *backdoor*, in this context, is a network route that bypasses security controls. Border routers of a monitored network should pass only those incoming packets that have source addresses outside of the monitored network's IP address space, and, unless they are transit networks, they should pass only those outgoing packets

[21] While this might be indicative of scanning activity, the task of scan detection is more complex than shown in Example 4.19. 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.

Figure 4.13: Growth Graph of Cumulative Number of Source IP Addresses by Hour



Example 4.20: Using `rwsettool` and `rwsetcat` to Track Server Usage

```
<1>$ for day in 20 21 22; do
        hr=0
        while [ $hr -lt 24 ]; do
           rwfilter --type=in --start-date=2009/04/${day}T$hr \
                    --daddress=192.168.200.6 --protocol=17 --dport=53 \
                    --pass=stdout \
             | rwset --sip-file=day-$day-hour-$hr.set
           let hr++
        done
     done
<2>$ cp day-20-hour-0.set buffer.set
<3>$ for fn in day-*.set; do
        rwsettool --union buffer.set $fn --output-path=newbuffer.set
        mv newbuffer.set buffer.set
        cnt=$(rwsetcat buffer.set --count-ips)
        echo "$fn $cnt" >>total_ips.txt
     done
<4>$ echo -e "\nDay/Hour Cumulative IPs"; head -n3 total_ips.txt

Day/Hour Cumulative IPs
day-20-hour-0.set 0
day-20-hour-10.set 0
day-20-hour-11.set 4
```

that have source addresses inside of that IP address space. However, a variety of routing anomalies and backdoors spoil this ideal.

The first step in performing backdoor analysis is actually defining the monitored network's IP address space. The easiest way to do this is to create a text file that describes the net blocks that make up the network using the `rwsetbuild` tool. In Example 4.21, when monitoring two address blocks, 192.168.1.0/24 and 192.168.2.0/24, the analyst created a text file called `mynetwork.set.txt` containing those two blocks on separate lines and then ran `rwsetbuild` to create a binary set file called `mynetwork.set`. Note that if the border router and flow sensor are outside of the Network Address Translation (NAT) device, the set should be built with global addresses (i.e., the translated addresses of the monitored network's hosts as they are known by the global Internet), not the monitored network's inside local (private) addresses.

Example 4.21: `rwsetbuild` for Building an Address Space IP Set

```
<1>$ cat <<END_FILE >mynetwork.set.txt \
192.168.1.0/24
192.168.2.0/24
END_FILE
<2>$ rwsetbuild mynetwork.set.txt mynetwork.set
```

Once the set exists, the analyst can use it as the basis for filtering. Malicious actors sometimes false-source traffic into the network using internal addresses, but that traffic is almost exclusively very short (only 1–3 packets) and has simple flag combinations. For example, filtering on source address identifies incoming

traffic from internal IP addresses as shown in Command 1 of Example 4.22, isolating traffic with content and indicators of complete TCP sessions. The inverse flows are much simpler, as shown in Command 2, since the border routers will only pass traffic destined for internal IP addresses to the Internet if the external routes are more efficient than the internal routes (and, depending on policy, perhaps the routers will never pass internal traffic externally). An analyst might also want to identify outgoing traffic originating from external IP addresses, as shown in Command 3. Such traffic indicates that the set of internal addresses is incomplete, the network is acting as a transit network, or an internal host is spoofing source addresses.

Example 4.22: Backdoor Filtering Based on Address Space

```
<1>$ rwfilter --sensor=SEN1 --type=in,inweb --start-date=2009/4/20T12 \
              --end-date=2009/4/20T13 --protocol=6 --bytes-per-packet=65- \
              --sipset=mynetwork.set --flags-all=SAF/SAFR,SAR/SAFR,SAFR/SAFR \
              --packets=4- --pass=strange_in.rw
<2>$ rwfilter --sensor=SEN1 --type=out,outweb --start-date=2009/4/20T12 \
              --end-date=2009/4/20T13 --dipset=mynetwork.set \
              --pass=strange_out.rw
<3>$ rwfilter --sensor=SEN1 --type=out,outweb --start-date=2009/4/20T12 \
              --end-date=2009/4/20T13 --not-sipset=mynetwork.set \
              --pass=strange_out2.rw
```

4.5 Summarizing Traffic with Bags

4.5.1 What Are Bags?

Bags are sets augmented with a volume measure for each key value and are generalized to accept several other flow aspects as well as addresses. Where IP sets record the presence or absence of particular 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 allow the analyst to summarize traffic on characteristics other than IP addresses—specifically on protocols and ports.[22]

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

4.5.2 Using `rwbag` to Generate Bags from Network Flow Data

As shown in Figure 4.14, `rwbag` generates files as specified by a group of parameters with a specific naming scheme: *`--key-count=filename`*. Each parameter of this group has a prefix that names the flow record field used as the key for the bag (source, destination, or next hop IP addresses; source or destination ports; router input or output interfaces; protocol number; or sensor). This is followed by a hyphen (`-`), and then a designation of which cumulative count to use as an aggregate value (flow records, packets, or bytes). Consequently, `rwbag --sip-flows` generates a bag file that counts flow records keyed on the source IP

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

address, `rwbag --dport-packets` generates a bag that counts packets keyed on the destination port, and `rwbag --proto-bytes` generates a bag that counts bytes keyed by protocol. The analyst may specify as many such parameters as required, although only one instance of each parameter is permitted per call. A sample invocation of `rwbag` is shown in Example 4.23.

Figure 4.14: Summary of `rwbag`

rwbag

Description	Generates bags from flow records
Call	`rwbag flow.rw --sip-bytes=x.bag --sip-flows=y.bag`
Parameters	***`--key-count`*** Writes bag of aggregated *count* by unique *key* value. May be specified multiple times. Allowed values for *key* and *count* are

Key		**Count**	
sip	Source IP address	flows	Count flow records
dip	Destination IP address	packets	Sum packets in records
nhip	Next-hop IP address	bytes	Sum bytes in records
input	Router input port		
output	Router output port		
sport	Source port		
dport	Destination port		
proto	Protocol		
sensor	Sensor ID		

For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.23: `rwbag` for Generating Bags

```
<1>$ rwfilter --sensor=SEN1 --type=in,inweb \
              --start-date=2009/4/20T12 --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
```

4.5.3 Using `rwbagbuild` to Generate Bags from IP Sets or Text

For data sources other than network flow records, the `rwbagbuild` tool creates bags from text or IP sets. Figure 4.15 provides a summary of `rwbagbuild` and its parameters. `rwbagbuild` supports several different uses:

- determine how many blacklists a given address is on
- determine how many time periods an address appears in network traffic
- for an address, count something other than bytes, packets, or flow records
- build an inverted bag, where the key is an aggregate and the value is a data element

Figure 4.15: Summary of `rwbagbuild`

rwbagbuild

Description	Creates a binary bag from non-flow data (expressed as text)
Call	`rwbagbuild --bag-input=ip-byte.bag.txt --key-type=sIPv4 --counter-type=sum-bytes --output-path=ip-byte.bag`
Parameters	Choose one: **`--set-input`** Creates a bag from the specified IP set, which may be `stdin` or a hyphen (`-`) **`--bag-input`** Creates a bag from a delimiter-separated text file, which can be `stdin` or a hyphen (`-`) Options: **`--delimiter`** Specifies the delimiter separating the key and value for the `--bag-input` parameter. Cannot be the pound sign (`#`) or the line-break character (new line) **`--default-count`** Specifies the integer count for each key in the new bag, overriding any values present in the input **`--key-type`** Sets the key type to this value. Allowable options are shown in Table 4.2. **`--counter-type`** Sets the counter type to this value. Allowable options are shown in Table 4.2. For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Table 4.2: `rwbagbuild` Key or Value Options

Type	Description, allowed values
`sIPv4`	Source IP addresses, IPv4 only, dotted quad, CIDR, wildcard, or integer
`dIPv4`	Destination IP address, IPv4 only, dotted quad, CIDR, wildcard, or integer
`sPort`	Source port, integer 0–65,535
`dPort`	Destination port, integer 0–65,535
`protocol`	IP protocol, integer 0–255
`packets`	Packet count, integer
`bytes`	Byte count, integer
`flags`	Bit string of TCP cumulative TCP flags (CEUAPRSF), integer 0–255
`sTime`	Starting time of the flow, integer seconds from UNIX epoch
`duration`	Duration of the flow, integer seconds
`eTime`	Ending time of the flow, integer seconds from UNIX epoch
`sensor`	Sensor ID, integer
`input`	SNMP index of input interface, integer
`output`	SNMP index of output interface, integer
`nhIPv4`	Next-hop IP address, IPv4 only, dotted quad, CIDR, wildcard, or integer
`initialFlags`	TCP flags in first packet in the flow, integer 0–255
`sessionFlags`	Bit string of cumulative TCP flags excluding the first packet, integer 0–255
`attributes`	Flags for termination conditions and packet size uniformity, integer
`application`	Guess as to the content of the flow, as set by the flow generator, integer 0–65,535
`class`	Class of the sensor, integer
`type`	Type of the flow, integer
`icmpTypeCode`	An encoded version of the ICMP type and code, where the type is in the upper byte and the code is in the lower byte
`sIPv6`	Source IP address, IPv6, canonical form or integer
`dIPv6`	Destination IP address, IPv6, canonical form or integer
`nhIPv6`	Next-hop IP address, IPv6, canonical form or integer
`records`	Count of flows, integer
`sum-packets`	Sum of packet counts, integer
`sum-bytes`	Sum of byte counts, integer
`sum-duration`	Sum of duration values, integer seconds
`any-IPv4`	Source, destination, or next-hop IPv4 address, dotted quad or integer
`any-IPv6`	Source, destination or next-hop IPv6 address, canonical form or integer
`any-port`	Source or destination port, integer 0–65,535
`any-snmp`	Input or output SNMP index of interface, integer
`any-time`	Start or end time value, integer seconds since UNIX epoch
`custom`	An integer

The input to `rwbagbuild` is either an IP set as specified in `--set-input` or a text file as specified in `--bag-input`, but not both. For IP-set 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 are allowed to be IP addresses (including canonical forms, CIDR blocks, and SiLK address wildcards) or unsigned integers. `rwbagbuild` 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, so times must be expressed as unsigned integer seconds since UNIX epoch. 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.

4.5.4 Reading Bags Using `rwbagcat`

`rwbagcat` reads bags to display or summarize their contents. This tool and its parameters are summarized in Figure 4.16. The default call to `rwbagcat` displays the contents of a bag in sorted order, as shown in Example 4.24.

Example 4.24: `rwbagcat` for Displaying Bags

```
<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|
```

In Example 4.24, the counts (the number of elements that match a particular IP address) are printed per key. `rwbagcat` provides additional display capabilities. For example, `rwbagcat` can print values within ranges of both counts and keys, as shown in Example 4.25.

These filtering values can be used in any combination. In addition to filtering, `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` command as shown in Example 4.26. In Command 1, this is shown using linear binning—one bin per value, with the counts showing how many keys had that value. In Command 2, this is shown with binary binning—values collected by powers of two and with counts of keys having aggregate volume values in those ranges. In Command 3, this 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 supports some logarithmic graphing options).

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

Figure 4.16: Summary of `rwbagcat`

rwbagcat

Description Displays or summarizes bag contents

Call `rwbagcat x.bag --output-path=x.bag.txt`

Parameters Choose one or none:

`--network-structure` Prints the sum of counters for each specified CIDR block in the comma-separated list of CIDR block sizes and/or letters (as described for `rwsetcat`; see Figure 4.11 on page 99)

`--bin-ips` Inverts the bag and counts keys by distinct aggregate volume values. Allowed arguments are `linear` (count keys with each volume [the default]), `binary` (count keys with volumes that fall in ranges based on powers of 2), and `decimal` (count keys that fall in ranges determined by a decimal logarithmic scale).

Options:

`--mincounter` Displays only entries with counts of at least the value given as the argument

`--maxcounter` Displays only entries with counts no larger than the value given as the argument

`--minkey` Displays only entries with keys of at least the value given as the argument

`--maxkey` Displays only entries with keys no larger than the value given as the argument

`--key-format` Specifies the formatting of keys for display. Allowed arguments are `canonical` (display keys as IP addresses in canonical format), `zero-padded` (display keys as IP addresses with zeroes added to fully fill width of column), `decimal` (display keys as decimal integer values), `hexadecimal` (display keys as hexadecimal integer values), and `force-ipv6` (display all keys as IP addresses in the canonical form for IPv6 with no IPv4 notation).

`--mask-set` Outputs records whose keys appear in the argument set

`--zero-counts` Prints keys with a counter of zero (requires `--mask-set` or both `--minkey` and `--maxkey`)

`--print-statistics` Prints statistics about the bag to given file or standard output

For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.25: `rwbagcat --mincounter`, `--maxcounter`, `--minkey`, and `--maxkey` to Filter Results

```
<1>$ rwbagcat ym.bag --mincounter=100 --maxcounter=105 \
      | head -n 5
  172.31.134.84|                100|
  172.31.194.55|                105|
 172.31.225.253|                102|
 172.31.234.255|                105|
 198.51.100.226|                102|
<2>$ rwbagcat ym.bag --minkey=172.31.2.0 --maxkey=172.31.5.0 \
      | head -n 5
     172.31.2.0|                  1|
     172.31.2.5|                 10|
     172.31.2.7|                  3|
    172.31.2.11|                  4|
    172.31.2.12|                 10|
```

Example 4.26: `rwbagcat --bin-ips` to Display Unique IP Addresses per Value

```
<1>$ rwbagcat ym.bag --bin-ips \
      | head -n 5
                  1|               2335|
                  2|               1850|
                  3|               1216|
                  4|                808|
                  5|                504|
<2>$ rwbagcat ym.bag --bin-ips=binary \
      | head -n 5
      2^00 to 2^01-1|               2335|
      2^01 to 2^02-1|               3066|
      2^02 to 2^03-1|               1811|
      2^03 to 2^04-1|               9304|
      2^04 to 2^05-1|                945|
<3>$ rwbagcat ym-big.bag --bin-ips=decimal \
      | head -n 5
                1011|                  1|
                1084|                  1|
                1364|                  2|
                1718|                  2|
                1758|                  1|
```

If the bag is not keyed by IP address, the `--key-format` parameter makes it *much* easier to read the output of `rwbagcat`. Example 4.27 shows the difference in output for a port-keyed bag counting bytes, where the larger port value is 65,000.

Example 4.27: `rwbagcat --key-format`

```
<1>$ rwbagcat in.bag
        0.0.0.3|                  56|
    0.0.253.232|                 280|
<2>$ rwbagcat in.bag --key-format=decimal
         3|                    56|
     65000|                   280|
<3>$ rwbagcat in.bag --key-format=hexadecimal
       3|                    56|
    fde8|                   280|
```

4.5.5 Manipulating Bags Using `rwbagtool`

`rwbagtool` provides bag manipulation capabilities (shown previously in Example 4.33), including adding and subtracting bags (analogous to the set operations), thresholding (filtering bags on volume), intersecting a bag and a set, and extracting a cover set from a bag. Figure 4.17 describes the syntax for `rwbagtool`.

Adding and Subtracting Bags

Because bags associate a count with each key value they contain, it is possible to perform arithmetic on bags, as well as performing threshold selection on the contents of a bag. The result is a bag with new volumes. To add bags together, use the `--add` parameter. The `--output-path` parameter specifies where to deposit the results. Most of the results from `rwbagtool` are bags themselves. Example 4.28 shows how bag addition works.

Example 4.28: Using `rwbagtool --add` to Merge Bags

```
<1>$ rwbagcat a.bag
     172.16.3.4|                  29|
    192.168.1.2|                  27|
<2>$ rwbagcat b.bag
      128.2.5.6|                  15|
     172.16.3.4|                  59|
    192.168.1.2|                  15|
<3>$ rwbagtool --add a.bag b.bag --output-path=c.bag
<4>$ rwbagcat c.bag
      128.2.5.6|                  15|
     172.16.3.4|                  88|
    192.168.1.2|                  42|
```

Figure 4.17: Summary of `rwbagtool`

rwbagtool

Description	Manipulates bags and generates cover sets
Call	`rwbagtool --add x.bag y.bag --output-path=z.bag`
Parameters	Choose one or none (with one input bag): **`--add`** Adds bags together (union) **`--subtract`** Subtracts from the first bag all the other bags (difference) **`--minimize`** Writes to the output the minimum counter for each key across all input bags **`--maximize`** Writes to the output the maximum counter for each key across all input bags **`--divide`** Divides the first bag by the second bag **`--scalar-multiply`** Multiplies each counter in the bag by the specified value. Accepts a single bag file as input. **`--compare`** Compares key/value pairs in exactly two bag files using the operation (OP) specified by the argument. Keeps only those keys in the first bag that also appear in the second bag and whose counter satisfies the OP relation with those in the second bag. The counter for each key that remains is set to 1. Allowed OPs are `lt` (less than), `le` (less than or equal to), `eq` (equal to), `ge` (greater than or equal to), `gt` (greater than). Choose zero or more masking/limiting parameters to restrict the results of the above operation or the sole input bag: **`--intersect`** Intersects the specified set with keys in the bag **`--complement-intersect`** Masks keys in the bag using IP addresses *not* in given IP-set file **`--minkey`** Cuts bag to entries with key of at least the value given as an argument **`--maxkey`** Cuts bag to entries with key of at most the value given as an argument **`--mincounter`** Outputs records whose counter is at least the value given as an argument, an integer **`--maxcounter`** Outputs records whose counter is not more than the value given as an argument, an integer Options: **`--coverset`** Generates an IP set for bag keys instead of creating a bag **`--invert`** Counts keys for each unique counter value **`--note-strip`** Does not copy notes from the input files to the output file **`--output-path`** Specifies where resulting bag or set should be stored For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

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, and any subtraction resulting in a negative number causes `rwbagtool` to omit the corresponding key from the resulting bag.

The multiplication operation on a bag is scalar multiplication: All the counter values in a bag are multiplied by the given scalar value. Example 4.29 provides an example using `--scalar-multiply=100` and `--divide` to provide percentages. Use bag division with care, as the second (denominator) bag must have every key found in the first (numerator) bag—one must not divide by zero. Often, this may require using the set intersection operation (discussed in the next section) to ensure that no mismatched elements are present.

Example 4.29: Using `rwbagtool` to Generate Percentages

```
<1>$ rwbagtool --scalar-multiply=100 b.bag --output-path=num.bag
<2>$ rwbagcat num.bag
      128.2.5.6|                1500|
     172.16.3.4|                5900|
    192.168.1.2|                1500|
<3>$ rwbagcat c.bag
      128.2.5.6|                  15|
     172.16.3.4|                  88|
    192.168.1.2|                  42|
<4>$ rwbagtool --divide num.bag c.bag --output-path=percent.bag
<5>$ rwbagcat percent.bag
      128.2.5.6|                 100|
     172.16.3.4|                  67|
    192.168.1.2|                  36|
```

Be careful when using the bag operations: Bags do have information in the file header about which types of keys and counts they contain, but the information is not used to restrict bag operations. Consequently, `rwbagtool` will add byte bags and packet bags together without warning, producing meaningless results. If unequal, but compatible, types are combined, 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), as shown with `rwfileinfo --fields=bag`.

Intersecting Bags and Sets

The `--intersect` and `--complement-intersect` parameters are used to intersect an IP set with a bag. Example 4.30 shows how to use these parameters to extract a specific subnet.

As Example 4.30 shows, `xf.bag` consists only of those IP addresses within the 198.51.x.x IP address block.

Isolating Parts of Bags with Count and Key Parameters

The same `--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 may be combined with an operation parameter (e.g., `--add`, `--subtract`) or a masking parameter (i.e., `--intersect` or `--complement-intersect`). As shown in Example 4.31, an analyst can combine thresholding with set intersection to get a bag holding only elements with keys in the set and values conforming to the threshold (at least five, in this example).

Example 4.30: Using `rwbagtool --intersect` to Extract a Subnet

```
<1>$ echo '198.51.x.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
    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|
  203.0.113.197|                   9|
<5>$ rwbagcat xf.bag
 198.51.100.227|                   3|
 198.51.100.244|                 101|
```

Example 4.31: `rwbagtool` Combining Threshold with Set Intersection

```
<1>$ rwbagtool xm.bag --intersect=f.set --mincounter=5 --output-path=xf2.bag
<2>$ rwbagcat xf2.bag | head -n 5
  198.51.225.158|              12|
   198.51.177.55|              51|
  198.51.188.134|             645|
  198.51.192.164|             740|
  198.51.224.164|              48|
```

Using `--coverset` to Extract Sets

Although bags cannot be used directly with `rwfilter`, the `--coverset` parameter can be used to obtain the set of IP addresses in a bag, and this set can be used with `rwfilter` and manipulated with any of the set commands. The `--coverset` parameter is used with the `--output-path` parameter, but in this case the result will be an IP set rather than a bag, as shown in Example 4.32.

Example 4.32: Using `rwbagtool --coverset` to Produce an IP Set from a Bag

```
<1>$ rwbagtool ym.bag --coverset --output-path=ym.set
<2>$ rwsetcat ym.set | head -n 3
172.31.0.2
172.31.0.3
172.31.0.4
<3>$ rwbagcat ym.bag | head -n 3
     172.31.0.2|                   2|
     172.31.0.3|                   7|
     172.31.0.4|                   1|
```

An analyst needs to be careful of bag content when using `--coverset`. Since `rwbagtool` does not limit operations with the type of keys bags contain, the `--coverset` parameter will interpret the keys as IP addresses even if they are actually protocol or port keys. This will likely lead to analysis errors.

4.5.6 Using Bags: A Scanning Example

To see how bags differ from sets in a useful way, this section revisits the scanning filter presented in Example 4.19 on page 104. The difficulty with that code is that if a scanner completed *any* handshake, it would be excluded from the `fast-only-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 4.33 is able to incorporate counts, resulting in the detection of more potential scanners. The calls to `rwfilter` in Commands 1 through 3 are piped to `rwbag` to build the initial bags (of incomplete, FIN-terminated and RST-terminated traffic, respectively). The latter two bags are merged in Command 4 to form a bag of completed connections. Commands 5 and 6 trim the complete- and incomplete-connection bags to the portions described above. Commands 7 and 8 generate the cover sets for these bags, and those cover sets are subtracted in Command 9, resulting in a scanning candidate set. The three sets are counted in Commands 10 through 12.

Example 4.33: Using `rwbag` to Filter Out a Set of Scanners

```
<1>$ rwfilter fastfile.rw --protocol=6 --flags-all=S/SRF --packets=1-3 \
                --pass=stdout \
      | rwbag --sip-flows=fast-low.bag
<2>$ rwfilter fastfile.rw --protocol=6 --flags-all=SAF/SARF,SR/SRF --pass=stdout \
      | rwbag --sip-flows=fast-high.bag
<3>$ rwbagtool fast-high.bag --maxcounter=10 --coverset --output-path=fast-high.set
<4>$ rwbagtool fast-low.bag --mincounter=10 --coverset --output-path=fast-low.set
<5>$ rwsettool --difference fast-low.set fast-high.set --output-path=scan.set
<6>$ rwsetcat fast-low.set fast-high.set scan.set --count-ips
fast-low.set:4
fast-high.set:1
scan.set:3
```

4.6 Labeling Flows with `rwgroup` and `rwmatch` to Indicate Relationship

`rwgroup` and `rwmatch` are grouping tools that allow an analyst to label a set of flow records that share common attributes with an identifier. This identifier, the group ID, is stored in the next-hop IP (`nhIP`) field, and it can be manipulated as an IP address (that is, either by directly specifying a group ID or by using IP sets).

The two tools generate group IDs in different ways. The `rwgroup` tool walks through a file of flow records and groups records that have common attributes, such as source or destination IP address pairs. The `rwmatch` tool 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.

For scalability purposes, the grouping tools require the data they process to 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).

4.6.1 Labeling Based on Common Attributes with `rwgroup`

The `rwgroup` tool provides a way to group flow records that have common field values. (See Figure 4.18 for a summary of this tool and its common parameters.) Once grouped, records in a group can be output separately (with each record in the group having a common ID) or summarized by a single record. Example applications of `rwgroup` include the following:

- grouping together all the 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 the flow records making 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.

Figure 4.18: Summary of `rwgroup`

rwgroup

Description	Flags flow records that have common attributes
Call	`rwgroup --id-fields=sIP --delta-field=sTime --delta-value=3600 --output-path=grouped.rw`
Parameters	Choose one or both: `--id-fields` Specifies the fields that need to be identical `--delta-field` Specifies the fields that need to be close. Requires the `--delta-value` parameter Options: `--delta-value` Specifies closeness for the `--delta-field` parameter `--objective` Specifies that the `--delta-value` argument applies relative to the first record, rather than the most recent `--rec-threshold` Specifies the minimum number of records in a group `--summarize` Produces a single record as output for each group, rather than all flow records For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

The criteria for a group are specified using the `--id-fields`, `--delta-field`, and `--delta-value` parameters; records are grouped when the fields specified by `--id-fields` are identical and the fields specified by `--delta-field` match within a value less than or equal to the value specified by `--delta-value`. Records in the same group will be assigned a common group ID. The output of `rwgroup` is a stream of flow records, where each record's next-hop IP address field is set to the value of the group ID. To do this grouping, `rwgroup` requires input records to be sorted by the fields specified in `--id-fields` and `--delta-field`.

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 4.34, a JABBER® session). To do so, the example sorts data on IP addresses and ports, and then groups together flow records that have closely related times. Prior to grouping, the example uses `rwsort` to sort all the fields that are specified to `rwgroup`.

`rwgroup`, by default, produces one flow record for every flow record it receives. Selective record production can be specified for `rwgroup` by using the `--rec-threshold` and `--summarize` parameters, as shown in Example 4.35. Using the `--rec-threshold` parameter specifies that `rwgroup` will only pass records that belong to a group with at least as many records as given in `--rec-threshold`. All other records will be dropped silently.

Example 4.35 shows how 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.

`rwgroup` can also generate summary records using the `--summarize` parameter. When this parameter is used, `rwgroup` will only produce a single record for each group; this record will use the first record in the

Example 4.34: Using rwgroup to Group Flows of a Long Session

```
<1>$ rwfilter --sensor=SEN1 --type=in,out --start-date=2009/4/15 \
             --end-date=2009/4/30 --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=0.0.0.5,12 --pass=stdout \
      | rwcut --fields=1-4,nhIP
            sIP|            dIP|sPort|dPort|          nhIP|
    172.16.20.6|   203.51.60.73|63820| 5222|        0.0.0.5|
    172.16.20.6|   203.51.60.73|63820| 5222|        0.0.0.5|
    172.16.20.6|   203.51.60.73|63820| 5222|        0.0.0.5|
    172.16.20.6|   203.51.60.73|63820| 5222|        0.0.0.5|
    172.16.20.6|   203.51.60.73|63820| 5222|        0.0.0.5|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
   203.51.60.73|    172.16.20.6| 5222|63820|       0.0.0.12|
```

Example 4.35: Using rwgroup --rec-threshold to Drop Trivial Groups

```
<1>$ rwsort flows.rw --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 \
      | rwcut --num-recs=10 --field=1-5,nhIP
            sIP|            dIP|sPort|dPort|pro|          nhIP|
    192.0.2.211| 192.168.130.16| 1177| 5222|  6|        0.0.0.0|
    192.0.2.211| 192.168.130.16| 1177| 5222|  6|        0.0.0.0|
    192.0.2.211| 192.168.130.16| 1177| 5222|  6|        0.0.0.0|
    192.0.2.211| 192.168.130.16| 1889| 5222|  6|        0.0.0.1|
 192.168.130.16|    192.0.2.211| 5222| 1177|  6|        0.0.0.2|
<3>$ 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|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 192.168.130.16| 198.51.100.161| 5222|63820|  6|        0.0.0.9|
 198.51.100.161| 192.168.130.16|63820| 5222|  6|       0.0.0.16|
 198.51.100.161| 192.168.130.16|63820| 5222|  6|       0.0.0.16|
 198.51.100.161| 192.168.130.16|63820| 5222|  6|       0.0.0.16|
 198.51.100.161| 192.168.130.16|63820| 5222|  6|       0.0.0.16|
```

group for its addressing information (IP addresses, ports, and protocol). The total number of bytes and packets for the group will be recorded in the summary record's corresponding fields, and the start and end times for the record will be the extrema for that group.[23] Example 4.36 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.

Example 4.36: Using `rwgroup --summarize` to Aggregate Groups

```
<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|
     192.0.2.64| 192.168.30.219|63820| 5222|  6|     34633|         0.0.0.5|
     192.0.2.64| 192.168.30.219|63820| 5222|  6|     26195|         0.0.0.5|
     192.0.2.64| 192.168.30.219|63820| 5222|  6|     10794|         0.0.0.5|
     192.0.2.64| 192.168.30.219|63820| 5222|  6|      6700|         0.0.0.5|
     192.0.2.64| 192.168.30.219|63820| 5222|  6|      6564|         0.0.0.5|
 192.168.30.219|     192.0.2.64| 5222|63820|  6|     67549|        0.0.0.12|
 192.168.30.219|     192.0.2.64| 5222|63820|  6|     26590|        0.0.0.12|
 192.168.30.219|     192.0.2.64| 5222|63820|  6|     10656|        0.0.0.12|
 192.168.30.219|     192.0.2.64| 5222|63820|  6|      8388|        0.0.0.12|
 192.168.30.219|     192.0.2.64| 5222|63820|  6|      5010|        0.0.0.12|
<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|
    192.0.2.186|192.168.122.251|63820| 5222|  6|     84886|         0.0.0.5|
192.168.122.251|    192.0.2.186| 5222|63820|  6|    123057|        0.0.0.12|
192.168.122.251|  198.51.100.49| 5222| 1177|  6|    275122|        0.0.0.14|
  198.51.100.49|192.168.122.251| 1177| 5222|  6|    101244|        0.0.0.16|
```

For any data file, calling `rwgroup` with the same `--id-fields` and `--delta-field` values will result in the same group IDs assigned to the same records. As a result, an analyst can use `rwgroup` to manipulate groups of flow records where the group has some specific attribute. This can be done using `rwgroup` and IP sets, as shown in Example 4.37. In Command 1, the analysis 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. Then (in Command 2), the analysis filters through the collection of groups for those group IDs (as next-hop IP addresses stored in `control.set`) that use FTP control. Finally (in Command 3), the analysis uses that next-hop IP set to pull out all the flows (ports 20 and 21) in groups that had FTP control (port 21) flows.

4.6.2 Labeling Matched Groups with `rwmatch`

`rwmatch` creates matched groups, where a matched group consists of an initial record (usually a *query*) followed by one or more *responses* and possibly additional queries. The calling syntax and some common options to `rwmatch` are shown in Figure 4.19. A response is a record that is related to the query (as specified in the `rwmatch` invocation) but is collected from a different direction or from a different router. As a result,

[23] 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.

Example 4.37: Using `rwgroup` to Identify Specific Sessions

```
<1>$ rwfilter flows.rw --protocol=6 --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|
 172.16.0.1|   198.3.0.2|59841|   21|  6|2009/04/20T14:49:07.047|
 172.16.0.1|   198.3.0.2|60031|   21|  6|2009/04/20T14:53:39.366|
 172.16.0.3|   198.3.0.4|19041|   21|  6|2009/04/20T14:35:40.885|
 203.51.0.5|   198.3.0.6| 1392|   21|  6|2009/04/20T13:56:03.271|
 203.51.0.5|   198.3.0.6| 1394|   21|  6|2009/04/20T13:56:04.657|
```

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.

The most basic use of `rwmatch` is to group records from both sides of a bidirectional session, such as HTTP requests and responses. However, `rwmatch` is capable of more flexible matching, such as across protocols to identify `traceroute` messages, which use UDP and ICMP.

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, and 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.

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.

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 4.38 shows the impact that not specifying all the fields has on TCP data. Note that the match relationship specified (query's source IP address matches response's destination IP address) results in all 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 in the second group shown).

Example 4.39 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. `rwmatch` is designed to handle not just protocols where source and destination are closely associated, but also where partial associations are significant.

Figure 4.19: Summary of `rwmatch`

rwmatch

Description	Matches IPv4 flow records that have stimulus-response relationships (IPv6 support introduced in Version 3.9.0)
Call	`rwmatch --relate=1,2 --relate=2,1 query.rw response.rw matched.rw`
Parameters	Choose one or none: **`--absolute-delta`** Includes potentially matching flows that start less than the interval specified by `--time-delta` after the end of the initial flow of the current match (default) **`--relative-delta`** Continues matching with flows that start within the interval specified by `--time-delta` from the greatest end time seen for previous members of the current match **`--infinite-delta`** After forming the initial pair of the match, continues matching on relate fields alone, ignoring time Options: **`--relate`** Specifies the numeric field IDs (1–8; see Figure 3.11 on page 59) that identify stimulus and response (required; may be specified multiple times) Starting with Version 3.9.0 values may be 1–8, 12–14, 20–21, 26–29, iType, and iCode; values may be specified by name or numeric ID. **`--time-delta`** Specifies the number of seconds by which a time window is extended beyond a record's end time. The default value is zero. **`--symmetric-delta`** Also makes an initial match for a query that starts between a response's start time and its end time extended by `--time-delta` **`--unmatched`** Includes unmatched records from the query file and/or the response file in the output. Allowed arguments (case-insensitive) are one of these: `q` (query file), `r` (response file), `b` (both) For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.38: Problem of Using `rwmatch` with Incomplete Relate Values

```
<1>$ rwfilter --sensor=SEN1 --type=in,out --start-date=2009/4/1 \
              --end-date=2009/4/30 --protocol=6 --dport=25 --pass=stdout \
      | rwsort --fields=1 --output-path=query.rw
<2>$ rwfilter --sensor=SEN1 --type=in,out --start-date=2009/4/1 \
              --end-date=2009/4/30 --protocol=6 --sport=25 --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|
    172.16.5.10|    198.51.1.25|45578|   25|2009/04/20T14:03:58.642|
    172.16.5.10|    198.51.1.25|45666|   25|2009/04/20T14:19:19.575|
    172.16.5.10|    198.51.1.25|45827|   25|2009/04/20T14:34:40.283|
    172.16.5.10|    198.51.1.25|45941|   25|2009/04/20T14:50:01.501|
<4>$ rwcut response.rw --fields=1-4,sTime --num-recs=4
            sIP|            dIP|sPort|dPort|                  sTime|
    198.51.1.25|    172.16.5.10|   25|13179|2009/04/23T16:07:38.550|
    198.51.1.25|    172.16.5.10|   25|13276|2009/04/23T16:11:39.339|
    198.51.1.25|    172.16.5.10|   25|17968|2009/04/24T13:12:31.226|
    198.51.1.25|    172.16.5.10|   25|18049|2009/04/24T13:21:03.343|
<5>$ rwmatch --relate=1,2 query.rw response.rw stdout \
      | rwcut --fields=1-4,nhIP --num-recs=5
            sIP|            dIP|sPort|dPort|           nhIP|
    172.16.5.10|    198.51.1.25|13179|   25|        0.0.0.1|
    198.51.1.25|    172.16.5.10|   25|13179|      255.0.0.1|
    172.16.5.10|    198.51.1.25|13276|   25|        0.0.0.2|
    172.16.5.10|    198.51.1.25|13109|   25|        0.0.0.2|
    198.51.1.25|    172.16.5.10|   25|13276|      255.0.0.2|
```

Example 4.39: Using `rwmatch` with Full TCP Fields

```
<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|
    172.16.5.10|    198.51.1.25|47767|   25|        0.0.0.1|
    198.51.1.25|    172.16.5.10|   25|47767|      255.0.0.1|
    172.16.5.10|    198.51.1.25|47851|   25|        0.0.0.2|
    198.51.1.25|    172.16.5.10|   25|47851|      255.0.0.2|
    172.16.5.10|    198.51.1.25|48576|   25|        0.0.0.3|
```

To establish a match group, first a response record must 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.

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. Queries that do not have a response and responses that do not have a query will be discarded unless `--unmatched` is specified. As a result, `rwmatch`'s output usually has fewer records than the total of the two source files. In Example 4.38, three flow records in the query file are discarded. `rwgroup` can be used to compensate for this by merging all the query records for a single session into one record.

Example 4.39 matches all addresses and ports in both directions. As with `rwgroup`, `rwmatch` requires sorted data, and in the case of `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 4.38 generated the query and response files from a query that might not produce sorted records; Example 4.39 corrected this by sorting the input files before calling `rwmatch`.)

`rwmatch` also can be used to match relationships across protocols. For example, traceroutes from UNIX hosts are generally initiated by a UDP call to port 33,434 and followed by an ICMPv4 "TTL exceeded" report message (type 11, code 0). A file of traces can then be composed by matching the ICMP responses to the UDP source as shown in Example 4.40. In this example, the key relationship is between the source of the query and the destination of the response. The source of the response may not be (and often, is not) the destination of the query. Therefore, only a single `--relate=1,2` is used.

Example 4.40: `rwmatch` for Matching Traceroutes

```
<1>$ rwfilter --start-date=2014/4/13 --type=in --sensor=SEN1 --protocol=17 \
              --dport=33434 --pass=stdout \
      | rwsort --field=1,sTime --output-path=queries.rw
<2>$ rwfilter --start-date=2014/4/13 --type=out --sensor=SEN1 --protocol=1 \
              --icmp-type=11 --icmp-code=0 --pass=stdout \
      | rwsort --field=2,sTime --output-path=responses.rw
<3>$ rwmatch --relate=1,2  queries.rw responses.rw traces.rw
```

4.7 Adding IP Attributes with Prefix Maps

Sometimes an analyst must associate a specific value to a range of IP addresses and filter or sort on the value rather than the address. One popular example is country codes: A common requirement would be to examine flow records associated with specific countries. An arbitrary association of addresses to labels is known as a prefix map (and abbreviated as pmap).

4.7.1 What Are Prefix Maps?

One type of SiLK prefix map defines an association between IP address ranges and text labels. Where an IP set creates a binary association between an address and a condition (an address is either in the set or not in the set), and a bag between an IP address and a value, a prefix map more generally assigns different values to many different address ranges. These arbitrary attributes can then be used in sorting, printing and filtering flow records.

To use prefix maps, an analyst must first create the pmap file itself. The analyst compiles a text-based mapping file containing the mapping of addresses to their labels and translates this text-based file into a binary pmap file. The pmap file can then be used by `rwfilter`, `rwcut`, `rwsort` and `rwuniq`.

4.7.2 Creating a Prefix Map

The tool `rwpmapbuild` creates binary prefix maps from text files. Each mapping line in the text file has an address range and a label to be assigned to the range, separated by whitespace (spaces and tabs). An address range can be specified with a whitespace-separated low IP address and high IP address (formatted either in canonical form or as integers), or with a CIDR block. A single host should be specified as a CIDR block with a prefix length of 32 for IPv4 or 128 for IPv6. A label may contain whitespace but no pound signs (`#`) and should not contain a comma (which would invalidate the pmap for use with `rwfilter`). The input file may also contain a default label to be applied to an address when there is no matching range. This default is specified by a line "`default` *deflabel*," where *deflabel* is the text label specified by the analyst for otherwise-unlabeled address ranges. The input file may specify a name for the pmap via a line "`map-name` *mapname*" where *mapname* is the desired name for this pmap, which cannot contain whitespace, commas, or colons. The common options for `rwpmapbuild` are summarized in Figure 4.20. Example 4.41 shows an example of creating a prefix map. This example of the use of prefix maps builds a map of suspected spyware distribution hosts.

4.7.3 Selecting Flow Records with `rwfilter` and Prefix Maps

There are three pmap parameters for `rwfilter`. `--pmap-file` specifies the compiled prefix map file to use and optionally associates a mapname with that pmap. `--pmap-src-`*mapname* and `--pmap-dst-`*mapname* specify the set of labels used for filtering records based on the source or destination addresses, respectively, where *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 4.42 shows the `rwfilter` parameters using the pmap built in Example 4.41 to select from a sample collection of flow records just those that come from spyware hosts.

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,

Figure 4.20: Summary of `rwpmapbuild`

rwpmapbuild

Description	Creates a prefix map from a text file
Call	`rwpmapbuild --input-file=sample.pmap.txt --output-file=sample.pmap`
Parameters	**`--input-file`** Specifies the text file that contains the mapping between addresses and prefixes. When omitted, read from `stdin` **`--mode`** Specifies the type of input as if a `mode` statement appeared in the input. Valid values are `ipv4`, `ipv6`, and `proto-port`. **`--output-file`** Specifies the filename for the binary prefix map file **`--ignore-errors`** Writes the output file despite any errors in the input For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.41: Using `rwpmapbuild` to Create a Spyware Pmap File

```
<1>$ cat <<END_FILE >spyware.pmap.txt
map-name spyware
default None

#  Spyware related address ranges
64.94.137.0/24              180solutions
205.205.86.0/24             180solutions
209.247.255.0/24            Alexa
209.237.237.0/24            Alexa
209.237.238.0/24            Alexa
216.52.17.12/32             BargainBuddy
198.65.220.221/32           Comet Cursor
64.94.162.0  64.94.162.100  Comet Cursor
64.94.89.0/27               Gator
64.162.206.0/25             Gator
82.137.0.0/16               Searchbar
END_FILE
<2>$ rwpmapbuild --input-file=spyware.pmap.txt --output-file=spyware.pmap
<3>$ file spyware.pmap.txt spyware.pmap
spyware.pmap.txt: ASCII text
spyware.pmap:     data
```

Example 4.42: Using Pmap Parameters with `rwfilter`

```
<1>$ rwfilter --sensor=SEN1 --type=in,out --start-date=2009/4/1 \
              --end-date=2009/4/30 --protocol=6 --pass=stdout \
      | rwfilter stdin --pmap-file=spyware.pmap \
                 --pmap-src-spyware=None --fail=spyware.rw
<2>$ rwfileinfo spyware.rw --fields=count-records
spyware.rw:
  count-records       484093
```

`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`.

4.7.4 Working with Prefix Values Using `rwcut` and `rwuniq`

To display the actual value of a prefix, `rwcut` can be used with the `--pmap-file` parameter, which 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`, but 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`. The `--pmap-file` parameter must precede the `--fields` parameter. The two pmap fields output labels associated with the source and destination IP addresses. Example 4.43 shows how to print out the type of spyware associated with an outbound flow record.

Example 4.43: Using `rwcut` with Prefix Maps

```
<1>$ rwcut spyware.rw --pmap-file=spyware.pmap \
           --fields=src-spyware,sPort,dIP,dPort --num-recs=5
 src-spyware|sPort|            dIP|dPort|
180solutions|   80|       10.0.0.1| 1132|
180solutions|   80|       10.0.0.1| 1137|
180solutions|   80|       10.0.0.2| 2746|
   Searchbar| 3406|       10.0.0.3|   25|
180solutions|   80|       10.0.0.2| 2746|
```

The `rwsort`, `rwgroup`, `rwstats`, and `rwuniq` tools also work with prefix maps. The prefix map parameters are the same as for `rwcut` and perform sorting, grouping, and counting operations as expected. Example 4.44 and Example 4.45 demonstrate using `rwsort` and `rwuniq` with prefix maps, respectively.

Example 4.44: Using `rwsort` with Prefix Maps

```
<1>$ rwsort spyware.rw --pmap-file=spyware.pmap --fields=src-spyware,bytes \
      | rwcut --pmap-file=spyware.pmap --fields=src-spyware,sport --num-recs=5
 src-spyware|sPort|
180solutions|   80|
180solutions|   80|
180solutions|   80|
180solutions|   80|
180solutions|   80|
```

4.7.5 Querying Prefix Map Labels with `rwpmaplookup`

When using prefix maps, an analyst will occasionally need to directly query the pmap, in addition to using it with other SiLK tools. `rwpmaplookup` provides a query capability for prefix maps, either user-defined pmaps

Example 4.45: Using `rwuniq` with Prefix Maps

```
<1>$ rwuniq spyware.rw --pmap-file=spyware.pmap --fields=src-spyware,dPort \
            --values=flows,dIP-Distinct \
      | head -n 5
 src-spyware|dPort|   Records|Unique_DIP|
180solutions| 1792|         4|         1|
   Searchbar| 3072|         6|         6|
   Searchbar|32512|         4|         1|
180solutions|64000|         2|         1|
```

or one of the two that often are created as part of the SiLK installation: country codes and address types. Figure 4.21 provides a summary of this tool's syntax and parameters. Address types have been previously discussed in conjunction with `rwcut`'s fields `sType` and `dType`. Country codes are used by the Root Zone Database (e.g., see http://www.iana.org/domains/root/db) and 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 lookup this information on the Internet.[24] Country code maps are not in the normal pmap binary format and cannot be built using `rwpmapbuild`.

Using `rwpmaplookup`, the analyst can query a list of addresses from a text file (the default), use `--ipset-files` to query from IP sets, or use `--no-files` to list the addresses 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. If the prefix map being queried is a protocol-and-port pmap, it makes no sense to query it with an IP set, and `rwpmaplookup` will print an error and exit if `--ipset-files` is given. For protocol-and-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. Example 4.46 shows examples of `rwpmaplookup`, where Command 1 identifies a list of addresses to look up, Command 2 looks up their country codes, Command 3 looks up their address types, Commands 4 and 5 build a protocol-port prefix map, and Command 6 looks up protocol/port pairs from the command line.

[24] 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.

Figure 4.21: Summary of `rwpmaplookup`

rwpmaplookup

Description	Shows label associated with an addresses' prefix map, address type, or country code
Call	`rwpmaplookup --map-file=spyware.map address-list.txt`
Parameters	Choose one: **`--map-file`** Specifies the pmap that contains the prefix map to query **`--address-types`** Finds IP addresses in the address-types mapping file specified in the argument or in the default file when no argument is provided **`--country-codes`** Finds IP addresses in the country-code mapping file specified in the argument or in the default file when no argument is provided Options: **`--fields`** Specifies the fields to print. Allowed values are: `key` (key used to query), `value` (label from prefix map), `input` (the text read from the input file [excluding comments] or IP address in canonical form from set input file), `block` (CIDR block containing the key), `start-block` (low address in CIDR block containing the key), and `end-block` (high address in CIDR block containing the key.) Default is `key,value`. **`--no-files`** Does not read from files and instead treat the command line arguments as the IP addresses or protocol/port pairs to find **`--no-errors`** Does not report errors parsing the input **`--ipset-files`** Treats the command-line arguments as names of binary IP-set files to read For additional parameters, see Table 4.5 on page 135 and Table 3.14 on page 72.

Example 4.46: Using `rwpmaplookup` to Query Addresses and Protocol/Ports

```
<1>$ cat <<END_FILE >ips_to_find
192.88.209.17
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 1/2303   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|
```

4.8 Gaining More Features with Plug-Ins

The SiLK tool suite is constantly expanding, with new tools and new features being added frequently. One of the ways that new features are being added is via dynamic library plug-ins for various tools. Table 4.3 provides a list of the current plug-ins distributed with SiLK.

Example 4.47 shows the use of a plug-in, in this case `cutmatch.so`. Once the plug-in is invoked using the `--plugin` parameter, it defines a `match` field, which formats the `rwmatch` results as shown.

Example 4.47: Using `rwcut` with `--plugin=cutmatch.so`

```
<1>$ rwcut matched.rw --plugin=cutmatch.so --fields=1,3,match,2,4,5
                 sIP|sPort| <->Match#|             dIP|dPort|pro|
     192.168.251.79|49636|->       1|    10.10.10.65|   80|  6|
        10.10.10.65|   80|<-       1| 192.168.251.79|49636|  6|
     192.168.251.79|49637|->       2|    10.10.10.65|   80|  6|
        10.10.10.65|   80|<-       2| 192.168.251.79|49637|  6|
```

Further documentation on these plug-ins is provided in Section 3 of *The SiLK Reference Guide* (http://tools.netsa.cert.org/silk/reference-guide.html).

Table 4.3: Current SiLK Plug-Ins

Name	Description
`cutmatch.so`	Lets `rwcut` present the `rwmatch` results in an easier-to-follow manner as `Match` field
`flowrate.so`	Provides `bytes/packet`, `bytes/second`, and `packets/second` fields to `rwcut`, `rwsort`, and `rwuniq`; adds `--bytes-per-second` and `--packets-per-second` parameters to `rwfilter`
`int-ext-fields.so`	Defines internal and external addresses and ports for `rwcut`, `rwgroup`, `rwsort`, `rwstats`, and `rwuniq`
`rwp2f_minbytes.so`	Allows minimum bytes or packet filtering with `rwptoflow`

4.9 Parameters Common to Several Commands

As with the essential SiLK tools, a number of parameters are used with the same semantics across groups of tools. This section describes this commonality in tabular form for the tools described in this chapter. Due to the breadth of tools covered, there are two tables, in which all the common parameters are listed, but the tools are split into two groups. These parameters are described at the end of Chapter 3. Table 3.14 on page 72 lists the same parameters as in Table 4.4 and Table 4.5.

Table 4.4: Common Parameters in Advanced SiLK Tools – Part 1

Parameter	**rwcat**	**rwappend**	**rwdedupe**	**rwsplit**	**rwfileinfo**	**rwtuc**	**rwptoflow**	**rwpmatch**	**rwnetmask**
`--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`									
`--pager`									
`--note-add`	✓		✓	✓		✓	✓		✓
`--note-file-add`	✓		✓	✓		✓	✓		✓
`--dry-run`									

For parameter descriptions, see Table 3.14 on page 72.

Table 4.5: Common Parameters in Advanced SiLK Tools – Part 2

Parameter	rwset	rwsetcat	rwsettool	rwsetbuild	rwbag	rwbagbuild	rwbagcat	rwbagtool	rwgroup	rwmatch	rwpmapbuild	rwpmaplookup
`--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`		✓										✓
`--pager`		✓					✓					✓
`--note-add`	✓		✓	✓	✓	✓		✓	✓	✓	✓	
`--note-file-add`	✓		✓	✓	✓	✓		✓	✓	✓	✓	
`--dry-run`											✓	

For parameter descriptions, see Table 3.14 on page 72.

Chapter 5

Using PySiLK for Advanced Analysis

This chapter presents how to use PySiLK to support several use cases that arise in analysis, but are difficult to implement within the normal constraints of the SiLK tool suite. Where appropriate, the analyst wishes to use portions of the SiLK suite functionality, but needs to use them 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 aggregate values for `rwuniq` and `rwstats`

Other 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 http://tools.netsa.cert.org/silk/reference-guide.pdf)
- detailed descriptions of the PySiLK structures, provided in *PySiLK: SiLK in Python* (http://tools.netsa.cert.org/silk/pysilk.pdf)
- larger PySiLK examples, provided in the PySiLK "tooltips" webpage (https://tools.netsa.cert.org/confluence/display/tt/Writing+PySiLK+scripts)
- 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/)

5.1 What Is PySiLK?

PySiLK is an extension to the SiLK tool suite that allows additional functionality expressed 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 built into SiLK, while leveraging those capabilities where appropriate.

Generally, to access PySiLK, the appropriate version of the Python language[25] must be available, and the PySiLK library (a directory named `silk`) should be loaded in the `site-packages` directory of the Python installation. To ensure the PySiLK library works properly, the `PYTHONPATH` environment variable should be set to include the `site-packages` directory.

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 by the CERT/CC as part of the PySiLK library. PySiLK provides the capability to manipulate SiLK objects (flow records, IP sets, 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), leveraging the processing of the SiLK tools by use of a plug-in will yield a more suitable solution than developing a stand-alone script.

PySiLK plug-ins for SiLK tools use an additional component, also provided by the CERT/CC, called `silkpython`. 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. For `rwstats` and `rwuniq`, two types of fields can be defined: *key* fields used to categorize flow records into bins and *aggregate value* fields used to compute a single value for each bin from the records in those bins. For all 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 aggregate value fields for `rwstats` and `rwuniq`. The advanced API provides one function that can create either key fields or aggregate value fields, and permits a higher degree of control over these fields.

5.2 Extending `rwfilter` with PySiLK

PySiLK extends the capabilities of `rwfilter` by letting the analyst create new methods for partitioning flow records.[26]

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.

[25] Python 2.7.x for SiLK Version 3.8.

[26] 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.

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 for `rwfilter` in these cases:

- Some information from prior records may help 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.

5.2.1 Using PySiLK to Incorporate State from Previous Records

For an example of where some information (or *state*) from prior records may help in partitioning subsequent records, consider Example 5.1. 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`.

In Example 5.1, 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 5.1, 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.

Example 5.1: `ThreeOrMore.py`: Using PySiLK for Memory in `rwfilter` Partitioning

```
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')
```

5.2.2 Using PySiLK with rwfilter in a Distributed or Multiprocessing Environment

An analyst could use a PySiLK script with `rwfilter` by first having a call to `rwfilter` that retrieves the records that satisfy a given set of conditions and then pipes those records to a second execution of `rwfilter` that uses the `--python-file` parameter to invoke the script. This is shown in Example 5.2. 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.

Example 5.2: Calling `ThreeOrMore.py`

```
<1>$ rwfilter --type=in --start-date=2010/8/27T13 --end-date=2010/8/27T22 \
              --protocol=6 --dport=25 --bytes-per-packet=65- --packets=4- \
              --flags-all=SAF/SAF,SAR/SAR --pass=stdout \
     | rwfilter stdin --python-file=ThreeOrMore.py --pass=email.rw
```

5.2.3 Simple PySiLK with rwfilter --python-expr

Some analyses that don't 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 5.3 partitions flow records that have the same port number for the source and the destination. Whereas the name for the flow record object is specified by the parameter name in user-written Python files, with `--python-expr`, the record object is always called `rec`.

Example 5.3: Using `--python-expr` for Partitioning

```
<1>$ rwfilter flows.rw --protocol=6,17 --python-expr='rec.sport==rec.dport' \
              --pass=equalports.rw
```

With `--python-expr`, it isn't possible to retain state from previous flow records as in Example 5.1. Nor is it possible to incorporate information from sources other than the flow records. Both of these require a plug-in invoked by `--python-file`.

5.2.4 PySiLK with Complex Combinations of Rules

Example 5.4 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. This would be even harder using a tuple file, since there is no equivalent in a tuple file for `rwfilter`'s `--aport` parameter.

Example 5.4: `vpn.py`: Using PySiLK with `rwfilter` for Partitioning Alternatives

```
def vpnfilter(rec):
  if (rec.protocol == 17  # UDP
    and (rec.dport in (500, 1194, 1224) or   # IKE, OpenVPN, VPNz
         rec.sport in (500, 1194, 1224)   ) ):
      return True
  if rec.protocol in (50, 51):  # ESP, AH
    return True
  return False

register_filter(vpnfilter)
```

5.2.5 Use of Data Structures in Partitioning

Example 5.5 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 5.5, called `matchblock.py`.

In Example 5.5, 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 Python 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 doesn't 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 wasn't specified in the blockfile), the test fails in line 38.

Example 5.6 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

Example 5.5: `matchblock.py`: Using PySiLK with `rwfilter` for Structured Conditions

```
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')
```

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.

Example 5.6: Calling `matchblock.py`

```
<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 --input-pipe=stdin --python-file=matchblock.py \
                 --blockfile=blockfile.csv --print-statistics
Files     1.  Read     375567.  Pass          8. Fail      375559.
```

5.3 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).

5.3.1 Computing Values from Multiple Records

Example 5.7 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. There are a number of potential uses for this, 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.

To use `delta.py`, Example 5.8 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.

5.3.2 Computing a Value Based on Multiple Fields in a Record

Example 5.9 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

Example 5.7: delta.py

```
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)
#                   --parameter     function    min    max
```

Example 5.8: Calling delta.py

```
<1>$ rwfilter --type=out --start-date=2010/8/30T00 \
              --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|
    172.28.7.88|    172.28.67.3|2010/08/30T00:05:09.909|         0|
    172.28.7.88| 172.28.130.136|2010/08/30T00:45:59.145|         0|
    172.28.7.88| 172.28.130.136|2010/08/30T00:47:01.282|     62137|
    172.28.7.88| 172.28.130.136|2010/08/30T00:52:13.168|    311885|
    172.28.7.88| 172.28.130.136|2010/08/30T00:57:25.270|    312101|
    172.28.7.88| 172.28.140.107|2010/08/30T00:15:05.989|         0|
    172.28.7.88|  172.28.142.28|2010/08/30T00:01:09.593|         0|
    172.28.7.88|  172.28.142.28|2010/08/30T00:01:31.732|     22139|
    172.28.7.88|  172.28.142.28|2010/08/30T00:03:39.565|    127832|
    172.28.7.88|  172.28.142.28|2010/08/30T00:04:51.700|     72134|
    172.28.7.88|  172.28.142.28|2010/08/30T00:07:43.104|    171404|
    172.28.7.88|  172.28.142.28|2010/08/30T00:08:11.665|     28560|
    172.28.7.88|  172.28.142.28|2010/08/30T00:09:04.014|     52348|
    172.28.7.88|  172.28.142.28|2010/08/30T00:09:29.517|     25503|
    172.28.7.88|  172.28.142.28|2010/08/30T00:09:30.359|       842|
    172.28.7.88|  172.28.142.28|2010/08/30T00:09:53.913|     23554|
    172.28.7.88|  172.28.142.28|2010/08/30T00:09:53.941|        27|
    172.28.7.88|  172.28.142.28|2010/08/30T00:10:08.274|     14332|
    172.28.7.88|  172.28.142.28|2010/08/30T00:13:24.160|    195886|
    172.28.7.88|  172.28.142.28|2010/08/30T00:15:14.318|    110157|
```

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.

Example 5.9: `payload.py`: Using PySiLK for Conditional Fields with `rwsort` and `rwcut`

```
#            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)
#                  --parameter   function  min       max
```

Example 5.10 shows how to use Example 5.9 with both `rwsort` and `rwcut`. The records are sorted into payload-size order and then output, showing both the bytes and payload values.

Example 5.10: Calling payload.py

```
<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|
```

5.4 Defining Key Fields and Aggregate 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 aggregate value fields. Key fields and aggregate fields use different registration functions in the simple API, however the registered callback functions don't have to be different. In Example 5.11, 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 aggregate value.

Example 5.11: bpp.py

```
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)
```

In Example 5.12, Command 2 uses the Python file, bpp.py, to create a key field for binning records. Command 3 creates an aggregate value field instead. The aggregate 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 still other aggregates, the analyst can use the advanced API function, register_field.

Example 5.12: Calling `bpp.py`

```
<1>$ rwfilter --type=in --start-date=2014/6/30T17 \
              --protocol=0- --max-pass-records=30 --pass=tmp.rw
<2>$ rwuniq tmp.rw --python-file=bpp.py --fields=protocol,bpp --values=records
pro|       bpp|   Records|
 17|        70|         1|
  6|       808|         1|
 17|       120|         1|
  6|       243|         1|
  6|       286|         1|
 17|        83|         2|
 17|        72|         6|
 17|        82|         4|
  6|        73|         1|
 17|        58|         1|
 17|        61|         2|
 17|        76|         2|
  6|        40|         3|
 17|        80|         1|
 17|        78|         1|
 17|        73|         1|
  6|       111|         1|
<3>$ rwuniq tmp.rw --python-file=bpp.py --fields=protocol --values=maxbpp
pro|    maxbpp|
  6|       808|
 17|       120|
```

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.

Chapter 6

Additional Information on SiLK

This handbook 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.

6.1 Contacting SiLK Support

The SiLK tool suite is available in open-source form at http://tools.netsa.cert.org/silk/. The CERT/CC also supports FloCon,® an annual workshop devoted to flow analysis. More information on FloCon is provided at http://www.cert.org/flocon/.

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 http://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 http://tools.netsa.cert.org/silk/reference-guide.html in HTML format and at http://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 http://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 http://tools.netsa.cert.org/silk/faq.html.

SiLK Tooltips: This wiki contains tips and tricks posted by clever SiLK users. This useful information often is not immediately obvious from the standard documentation. Find it at https://tools.netsa.cert.org/tooltips.html.

PySiLK Tooltips: Tips for writing PySiLK scripts are provided at https://tools.netsa.cert.org/confluence/display/tt/Writing+PySiLK+scripts.

The primary SiLK mailing 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/CC technical reports to reach the reports' authors. The focus is on analytical techniques. Public reports are provided at http://www.cert.org/netsa/publications/.

flocommunity@cert.org: This distribution list addresses a community of analysts built on the core of the FloCon conference (http://www.cert.org/flocon/). The initial focus is on flow-based network analysis, but the scope will likely naturally expand to cover other areas of network security analysis. The list is not focused exclusively on FloCon itself, although it will include announcements of FloCon events.

The general philosophy of this email list and site is inclusive: The CERT/CC intends to include international participants from both research and operational environments. Participants may come from universities, corporations, government entities, and contractors. Additional information is accessible via the FloCommunity webpage (http://www.cert.org/flocon/flocommunity/).

Since the FloCon conference covers a range of network security topics, especially network flow analysis, the conference organizers encourage ongoing discussions. In support of this, the following social networking opportunities are offered:

"FloCon Conference" LinkedIn member: The FloCon Conference member page (http://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 http://www.linkedin.com/groups?gid=3636774. Here, members discuss matters related to the FloCon conference and network flow analysis.

@FloCon2015 Twitter account: The FloCon conference organizers post notices related to FloCon here. View (and follow!) the FloCon tweets at https://twitter.com/FloCon2015. The next sequential Twitter handle will be used for conferences in subsequent years.

Using SiLK for Network Traffic Analysis

Analyst's Handbook for SiLK Versions 3.8.3 and Later

October 2014

CERT Coordination Center®