A package for capturing and analyzing network data features

Joy version 2.0
January 2018

Preface

Joy is a BSD-licensed open source software package for collecting and analyzing network data, with a focus on network data feature exploration. This document shows how it can be used, installed, built, and modified. We hope that you find it useful, and that you enjoy using it. Please do understand that it is open source software, with the following licensing terms:

We gratefully acknowledge the support of our employer for the development of this package, and to the people who contributed to it, including Ellie Daw and Luke Valenta.

– the Joy team: Philip Perricone, Bill Hudson, Blake Anderson, Brian Long, and David McGrew

Contents

Chapter 1

Introduction

1.1 Overview

The Joy package contains a data collection program, `joy`, and some data analysis programs, including `sleuth`. The former is written in C99, and it reads raw network traffic or a packet capture file, and then outputs a summary of the observed traffic in JavaScript Object Notation (JSON) [**?**] format; see Figure **??** for an example. That program can also operate in telemetry collector mode, and receive IPFIX or Netflow version 9 packets; in that mode, it translates the telemetry into a JSON description of the flows that are observed by the telemetry exporters. The `sleuth` program reads a JSON description of traffic, filters the traffic as specified on its command line, and prints out the description of selected traffic, or it can select, summarize, or analyze traffic, as described in Section **??**.

1.1.1 Why Joy?

Joy is useful for collecting, analyzing, and exploring network data. Its JSON output formats are flexible and well-suited for many data analysis tools and modern programming environments, such as `python` and `scikit-learn`, and it uses a flow-oriented model that is both convenient and conceptually familiar. It can obtain network metadata while preserving privacy, by avoiding bulk data capture and by anonymizing addresses and usernames. It is aware of several important protocols, including HTTP, TLS, DNS, and DHCP, in the sense of being able to record their session metadata elements in its JSON format. It is also easily extensible; JSON gracefully handles optional or additional data, and the capture tool has a C preprocessor interface facilitating the capture of new data elements.

Joy does not know about as many protocols as do `wireshark`, `tshark`, or `tcpdump`, so the latter tools may be better suited for a detailed human-driven forensic analysis of a particular packet capture file. However, Joy is better suited for analyzing many sessions or many packet capture files.

Flow, in positive psychology, is a state in which a person performing an activity is fully immersed in a feeling of energized focus, deep involvement, and joy. This second meaning inspired the choice of name for this software package.

1.1.2 Ethics

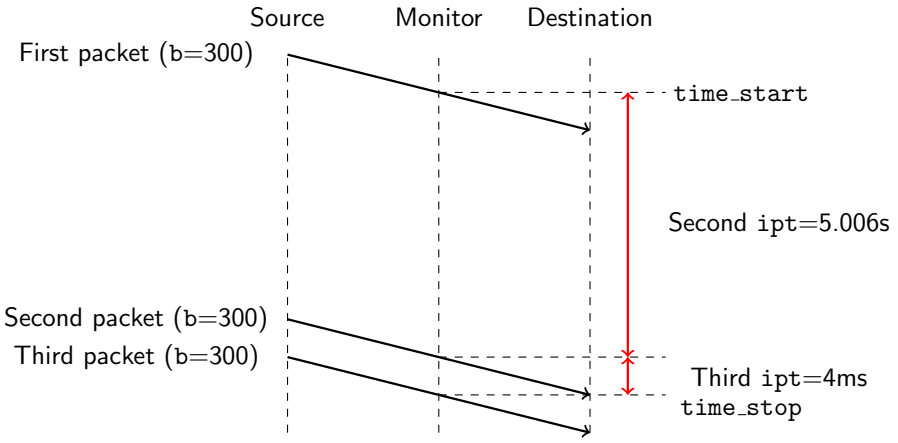
Joy is intended for use in network and security research, forensics, and for the monitoring of (small scale) networks to detect vulnerabilities, threats and other unauthorized or unwanted behavior. Researchers, administrators, penetration testers, and security operations teams can put this information to good use, for the protection of the networks being monitored, and in the case of vulnerabilities, for the benefit of the broader community through improved defensive posture. As with any network monitoring tool, Joy could potentially be misused; **do not use it on any network of which you are not the owner or the administrator**.

Figure 1.1: An example JSON description of a unidirectional flow. See Figure **??** for an illustration of this flow.

.

```
{
  "sa": "0.0.0.0",                    // IP source address
  "da": "255.255.255.255",            // IP destination address
  "pr": 17,                           // IP protocol number (17 = UDP)
  "sp": 68,                           // UDP source port
  "dp": 67,                           // UDP destination port
  "bytes_out": 900,                   // bytes sent from sa to da
  "num_pkts_out": 3,                  // packets sent from sa to da
  "time_start": 1479227824.653818,    // start time in seconds since the epoch
  "time_end": 1479227829.665744,      // end time in seconds since the epoch
  "packets": [                        // array of packet information
    {
      "b": 300,                       // bytes in UDP Data field
      "dir": ">",                     // direction: sa -> da
      "ipt": 0                        // inter-packet time: 0 ms since time_start
    },
    {
      "b": 300,                       // bytes in UDP Data field
      "dir": ">",                     // direction: sa -> da
      "ipt": 5006                     // inter-packet time: 5006 ms since last packet
    },
    {
      "b": 300,                       // bytes in UDP Data field
      "dir": ">",                     // direction: sa -> da
      "ipt": 4                        // inter-packet time: 4 ms since last packet
    }
  ],
  "ip": {
    "out": {
      "ttl": 128,
      "id": [
        1,
        2,
        3
      ]
    }
  },
  "expire_type": "i"
}
```

Figure 1.2: An illustration of the example unidirectional flow from Figure **??**, showing the relationship between the `start_time`, `stop_time`, and the `ipt` (inter-packet time) values. The first `ipt` value is zero.



1.2 Network data background

A *flow* is a set of packets with common characteristics, called a *flow key*. In Joy, the flow key is the conventional network five-tuple: the source and destination addresses, the source and destination port (for TCP and UDP traffic) and the protocol number. Symbolically, a flow key is the tuple (`sa, da, pr, sp, dp`). As defined above, a flow is *unidirectional*; all of its packets move in the same direction. Interactive network sessions usually involve *bidirectional* flows, which are described in Section **??**.

In this document, we sometimes use the term flow to mean a data record that describes a flow; this convention is commonplace in computer networking. The following table shows the flow key fields in the JSON format that Joy uses, along with some other important ones:

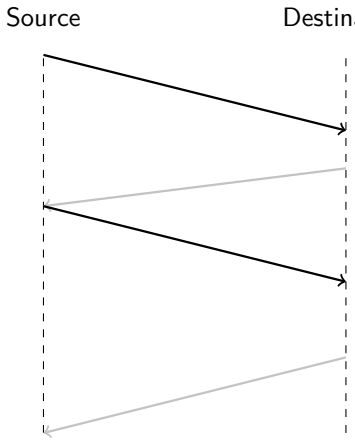
JSON element	Data element	Notes
`sa`	Source Address	Internet Protocol
`da`	Destination Address	
`pr`	Protocol Number	
`sp`	Source Port	TCP or UDP
`dp`	Destination Port	
`bytes_out`	Number of bytes sa→da in TCP, UDP, ICMP, or IP Data fields	Any Protocol
`num_pkts_out`	Number of packets sa→da	
`time_start`	Start time, seconds since epoch	
`time_end`	End time, seconds since epoch	
`packets`	Array of packet lengths, directions, and times	
`bytes_in`	Number of bytes sa←da in TCP, UDP, ICMP, or IP Data fields	Bidirectional flows only
`num_pkts_in`	Number of packets sa←da	

The epoch is defined by POSIX as 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970.

By default, `joy` does **not** report TCP packets with a zero-length Data field in the `packets` array. All TCP sessions start out with a handshake that contains two such packets, and most TCP sessions include 'ACK packets' that contain no data, and serve only to acknowledge the receipt of data sent by the other side. See Section **??** for more information.

Figure 1.3: Bidirectional flow examples.

(a) The correspondence between a bidirectional flow and its component outbound (black) and inbound (grey) unidirectional flows.



(b) A JSON example of a bidirectional flow, noting inbound data elements.

```
{
  "sa": "172.16.87.100",
  "da": "52.72.161.26",
  "pr": 6,
  "sp": 49173,
  "dp": 443,
  "bytes_out": 338,
  "num_pkts_out": 7,
  "bytes_in": 5629,      // inbound
  "num_pkts_in": 9,      // inbound
  "time_start": 1479227853.446511,
  "time_end": 1479227854.969042,
  ...
}
```

Joy extracts protocol-specific metadata, such as HTTP headers, TLS certificates, or DNS request names, and stores the data for each protocol in a JSON sub-object, such as `http`, `tls`, `dns`. To capture data for a particular protocol, the appropriate command line option must be passed to `joy`, as described in Section **??**. To select particular data elements from each flow, the `sleuth --select` facility can be used, as described in Section **??**.

1.2.1 Bidirectional flows

By default, `joy` captures unidirectional flows, but in many cases, bidirectional flows are more interesting; the `bidir=1` option (Section **??**) causes that program to stitch together any unidirectional flows that are part of the same session. A *bidirectional* flow consists of a pair of unidirectional flows whose source and destination addresses and ports are reversed, and whose time spans overlap. That is, if the flows `out` and `in` are unidirectional, then their combination is a bidirectional flow when

$$\begin{aligned} \texttt{in.sa} &= \texttt{out.da}, \\ \texttt{in.da} &= \texttt{out.sa}, \\ \texttt{in.sp} &= \texttt{out.dp}, \\ \texttt{in.dp} &= \texttt{out.sp}, \\ \texttt{in.pr} &= \texttt{out.pr}, \\ \texttt{in.start\_time} &> \texttt{out.start\_time}, \\ \texttt{in.start\_time} &< \texttt{out.stop\_time}. \end{aligned} \tag{1.1}$$

Figure **??** shows the relationship between two unidirectional flows and their bidirectional combination, and Figure **??** shows an example Joy output for a bidirectional flow. Joy uses the convention that, in a bidirectional flow, the packets flowing from `sa` (source) to `da` (destination) are *outbound*, and the packets flowing from `da` to `sa` are *inbound*; whenever `out` and `in` are used in the JSON schema, this directionality is implied.

A bidirectional flow is sometimes called a *biflow*.

1.2.2 Flow expiration

It is undesirable, in network monitoring, to wait for a long duration flow to finish before its flow record becomes available. Some flows last nearly as long as the uptime of the operating systems to which they connect. To avoid this indefinite wait for flow data, monitoring systems use a *timeout* approach: a flow

record is exported whenever a flow is inactive for an *inactive timeout* period, or whenever it is active and its duration exceeds an *active timeout* period. The `joy` capture tool uses these conventions, with an inactive timeout period of ten seconds, and an active timeout period of thirty seconds. That is, a flow record will be created if a flow is inactive for ten seconds, or if it is active for thirty seconds. The flow record indicates which of these conditions happened, in the `expire_type` element, which appears in the top level of each flow object:

- `"expire_type": "i"` denotes an inactive expiration, and
- `"expire_type": "a"` denotes an active expiration.

The `sleuth` program by default stitches together successive flows that have matching flow keys. Chronological stitching can be turned off (see Section **??**), which can be useful to understand how the timeout logic of Netflow or IPFIX monitoring systems affects data capture. However, in most cases, chronological stitching should be used. This is especially true when monitoring a private network, because flows that undergo active timeouts **can appear to originate outside the private network**. More to the point: a lack of chronological stitching can make it appear that there are flows that violate a private network access control policy. The TCP SYN flag can be used as a sanity check, since a long-running TCP flow that is truncated into two more flow records by an active timeout will only contain a SYN flag in the first flow record.

Chapter 2

The `joy` tool

The command line syntax to invoke joy is

```
joy [options] [file1 [file2 ... ]]
```

By default, its output is GZIP compressed JSON, using the format described in Section **??**. Uncompressed output, or bz2 compressed output, are available as compile-time options (see the file `src/include/output.h`). There are many software tools that work with GZIP compressed data, especially on Linux and other POSIX environments, such as `gunzip`, `zless`, and `zcat`. An example invocation that applies `joy` in offline mode and uses `gunzip` to convert its output to uncompressed form:

```
joy bidir=1 browse.pcap | gunzip

{"version":"1.74","interface":"none","promisc":0,"output":"none", ... }
{"sa":"10.0.2.15","da":"74.125.228.207","pr":6,"sp":43039,"dp":443, ... }
{"sa":"10.0.2.15","da":"74.125.228.195","pr":6,"sp":54210,"dp":443, ... }
{"sa":"10.0.2.15","da":"74.125.228.104","pr":6,"sp":47443,"dp":443, ... }
...
```

2.1 Overview

Each `joy` option (except `-x`) has the form `option=value`, where `option` is a string that identifies the option, and `value` is either a boolean (`0` or `1`), a (nonnegative) number, or a character string. There are two types of options: general ones (**??**) and ones that control data features that are output (**??**).

Joy can be run in several different modes. In **offline mode**, joy processes one or more packet capture (PCAP) files and prints a (compressed) JSON summary of the network flows to `stdout`; this mode is indicated by one or more file names on the command line. The file names *must* follow any options that are present, and each file name *should not* contain the equals sign (=), to avoid the possibility that joy would confuse a PCAP file with an option.

In **online mode**, joy listens to one or more network interfaces. This mode is indicated by using the `interface=I` option (Section **??**) to specify the interface `I`. If the option `output=F` (Section **??**) is present, then the (compressed) JSON output is written to file `F`. In this mode, output file rotation can be specified with `count` (Section **??**), and a log file can be specified with `logfile` (Section **??**). Output files can be uploaded to a separate server via SCP with the `upload` option (Section **??**). Online mode requires `root` privileges on most operating systems; for security, `joy` will drop its privileges after starting a capture (Section **??**).

Joy can also be used in Netflow or IPFIX **collector mode** (Sections **??**, **??**) or IPFIX **exporter mode** (Sections **??**, **??**, and **??**).

2.1.1 Configuration object

At initialization and before any other output, joy writes out a JSON object that describes its complete configuration, and the version of the joy program that produced it. This information is important, because the options affect what data elements can appear in the flow objects. Programs that analyze the joy JSON output *should* check for the configuration object and make sure that they do not process it as though it were a flow object, and they *should* do this by checking for the presence of the version field at the top level of the object. An annotated example of a configuration object:

```
{
  "version":"1.74",       // version of joy program
  "interface":"none",     // no output interface specified
  "promisc":0,            // no promiscuous mode
  "output":"none",        // no output specified; stdout will be used
  "outputdir":"none",     // no output directory specified
  "username":"none",      // no username specified for privilege dropping
  ...
}
```

The JSON names in the configuration object are joy command names, as defined in this chapter.

2.1.2 Subnet files

Some options cause joy to read a set of subnets from a file, and perform specific processing when sa and/or da are in one of those subnets. The format of a subnet file is illustrated by file internal.net:

```
# subnets for address anonymization
#

10.0.0.0/8          #  RFC 1918 address space
172.16.0.0/12       #  RFC 1918 address space
192.168.0.0/16      #  RFC 1918 address space
```

Each line contains an address as a dotted quad, or a subnet as a dotted quad followed by a slash and a number between 0 and 32, or is empty. A # character, and any characters on a line following it, are ignored, as are empty lines.

2.2 General options

2.2.1 -x F (string)

> Syntax:
>
> -x F

The command -x F causes joy to read configuration commands from the file F.

2.2.2 interface=I (string)

> Syntax:
>
> `interface=I`

The command `interface=I` causes `joy` to read packets live from network interface `I`, in online mode.

2.2.3 promisc=1 (boolean)

> Syntax:
>
> `promisc=1`

If `promisc=1`, and running in online mode, put the network interface into promiscuous mode.

2.2.4 output=F (string)

> Syntax:
>
> `output=F`

Write output to file `F`; otherwise `stdout` is used.

2.2.5 logfile=F (string)

> `logfile=F`

Write secondary output to file `F`; otherwise `stderr` is used. Secondary output consists of status reports, warnings, and errors.

2.2.6 count=C (number)

> Syntax:
>
> `count=C`

Rotate output files so each contains about `C` records.

2.2.7 upload=userserver:path (string)

> `upload=user@server:path`

Upload data files using the secure copy utility `scp` to the location `userserver:path`, after file rotation.

2.2.8 keyfile=F (string)

> Syntax:
>
> `keyfile=F`

When uploading data files using the `upload` command (Section **??**), use the SSH identity (private key) in file `F` to authenticate to the server.

2.2.9 anon=F (string)

> Syntax:
>
> `anon=F`

Anonymize addresses matching the subnets listed in file `F`. The format of the subnet file is described in Section **??**.

2.2.10 retain=1 (boolean)

> Syntax:
>
> `retain=1`

If `retain=1`, retain a local copy of each data file after it is uploaded.

2.2.11 preemptive_timeout=1 (boolean)

> Syntax:
>
> `preemptive_timeout=1`

If `preemptive_timeout=1`, then use active flow timeout logic that uses the packet timestamp to decide if adding that packet to the flow record would trigger a timeout.

2.2.12 nfv9_port=N (number)

> Syntax:
>
> `nfv9_port=N`

If `nfv9_port=1`, enable Netflow V9 capture on port `N`. Netflow v9 [**?**] reports basic flow telemetry.

2.2.13 ipfix_collect_port=N (number)

> Syntax:
>
> `ipfix_collect_port=N`

If `ipfix_collect_port=1`, enable IPFIX capture on port `N`. IPFIX [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] is an Internet Engineering Task Force (IETF) standard protocol for flow information export.

2.2.14 ipfix_collect_online=1 (boolean)

> Syntax:
>
> `ipfix_collect_online=1`

If `ipfix_collect_online=1`, have the IPFIX collector listen on a UDP socket.

2.2.15 ipfix_export_port=N (number)

> Syntax:
>
> `ipfix_export_port=N`

If `ipfix_export_port=N` is set, enable IPFIX export on port `N`.

2.2.16 ipfix_export_remote_port=N (number)

> Syntax:
>
> `ipfix_export_remote_port=N`

If `ipfix_export_remote_port=N` is set, then the IPFIX exporter will send to port `N` on the remote collector. Otherwise, the default port of 4739 is used.

2.2.17 ipfix_export_remote_host=host (string)

> Syntax:
>
> `ipfix_export_remote_host=host`

If `ipfix_export_remote_host=host` is set, then use `host` as the remote server target for the IPFIX exporter. Otherwise, the default `127.0.0.1` (localhost) address is used.

2.2.18 ipfix_export_template=type (string)

> Syntax:
>
> `ipfix_export_template=type`

If `ipfix_export_template=type` is set, then use `type` as the template for IPFIX exporter. The available types are `simple` and `idp`, and the default is the former.

2.2.19 aux_resource_path=path (string)

> Syntax:
>
> `aux_resource_path=path`

If `aux_resource_path=path` is set, then expect the auxiliary data files to be stored in the directory `path`.

2.2.20 verbosity=L (boolean)

> `verbosity=L`

If `verbosity=L)` is set, then the log (secondary) output includes events at level `L` and higher, where 0=off, 1=debug, 2=info, 3=warning, 4=error, 5=critical. The default value is 4 (error).

2.2.21 show_config=1

> Syntax:
>
> `show_config=1`

If `show_config=1`, then show the configuration on `stderr` when the program is started.

2.2.22 show_interfaces=1

> Syntax:
>
> `show_interfaces=1`

If `show_interfaces=1`, then show the interfaces on `stderr` when the program is started.

2.2.23 username=user

> Syntax:
>
> `username=user`

If `username=user` is set, then drop privileges to username `user` after starting packet capture in online mode as `root`. Otherwise, the default username `joy` is used.

2.3 Data feature options

2.3.1 bpf=expression

> Syntax:
>
> `bpf="expression"`

Filter all traffic by applying the Berkeley/BSD Packet Filter [**?**] (BPF) expression indicated on the command line, and only report flows matching the expression. Examples of BPF include `tcp` to capture only TCP traffic, and `udp port 53` to capture conventional DNS traffic.

2.3.2 zeros=1 (boolean)

> Syntax:
>
> `zeros=1`

Include packets with zero-length Data fields (e.g. TCP ACKs) in the `packets` array.

2.3.3 retrans=1 (boolean)

> Syntax:
>
> `retrans=1`

Include TCP retransmissions in the `packets` array.

2.3.4 bidir=1 (boolean)

> Syntax:
>
> `bidir=1`

Merge overlapping unidirectional flows into bidirectional flows. See Section **??** for background.

2.3.5 dist=1 (boolean)

> Syntax:
>
> `dist=1`

Include the `byte_dist`, or byte distribution, array as a top-level element in the JSON flow object. The array contains a count of the number of occurrences of each byte value in the Data fields of a flow, with the n^{th} element of the array corresponding to the byte value n. For instance, in the following example, the count of the byte 0 (hex 00) is 168, the count of byte 1 (hex 01) is 182, and the counts of bytes 16 (hex 10) and 255 (hex FF) are 81 and 77, respectively.


```
"byte_dist": [
    168, 182, 102, 190, 153, 103, 153,  80,
     75,  82,  86,  95,  80,  80,  83,  82,
     81,  73,  82,  94,  94,  52,  93,  83,
     66,  67,  68,  68,  85,  97,  70,  90,
     83,  75,  61,  98,  69,  68,  62,  54,
     67,  67, 117,  78,  59,  68, 195,  97,
    192, 112,  73,  71,  75,  70,  67,  63,
     69,  81,  73,  63,  70,  76,  60,  62,
     64,  68,  71,  68,  58,  67,  69,  91,
     70,  86,  70,  61,  56,  60,  74,  71,
     79,  55,  67,  85,  75, 121,  62,  52,
     71,  63,  77,  67,  72,  69,  68,  59,
     63,  95,  66, 138,  79, 150,  78, 148,
     83,  84,  78,  78, 119, 121, 101, 232,
     95,  84,  95,  91, 123, 107,  58,  64,
     68,  89,  63,  68,  59,  61,  71,  53,
     73,  59, 148,  62,  61,  63,  83,  55,
     66,  64,  61,  58,  61,  48,  61,  60,
     64,  83,  74,  64,  59,  66,  54,  63,
     78,  60,  59,  65,  81,  71,  70,  67,
     77,  65,  67,  73,  77,  59,  68,  67,
     57,  56,  63,  68,  56,  65,  61,  74,
     65,  68,  68,  76,  71,  61,  69,  61,
     66,  74,  67,  62,  68,  65,  68,  74,
     84,  59,  74,  59,  73,  73,  61,  63,
     87,  69,  69,  77,  58,  83,  65,  65,
     68,  61,  52,  63,  64,  78,  63,  64,
     74,  59,  61,  80,  60,  56,  55,  69,
     68,  82,  61,  55,  63,  61,  81,  41,
     63,  76,  64,  69,  67,  65,  72,  75,
     83,  60,  59,  72,  83,  62,  79,  61,
     74,  59,  63,  88,  68,  62,  66,  77
  ]
```

2.3.6 cdist=F (string)

> Syntax:
>
> `cdist=F`

Include compact byte distribution array using the mapping file F given in the command.

2.3.7 entropy=1 (boolean)

> Syntax:
>
> `entropy=1`

Report the entropy of the Data fields of the flow. This option causes the entropy to be reported as

- `entropy` is the entropy in bits per byte; this is the empirical entropy computed from the empirical probability distribution over the bytes. This number ranges from zero to 8.
- `total_entropy` is total entropy, in bytes, over all of the bytes in the flow. This number ranges from zero to $8n$, where n is `bytes_out` for unidirectional flows, and is `bytes_out` plus `bytes_in` for bidirectional flows.

An example follows.

```
{
  "entropy": 7.224162
  "total_entropy": 463054.342398,
}
```

2.3.8 exe=1 (boolean)

> Syntax:
>
> `exe=1`

If `exe=1`, include information about host process associated with flow. This information is only available when `joy` is run on the host for that process.

2.3.9 classify=1 (boolean)

> Syntax:
>
> `classify=1`

If `classify=1`, include results of post-collection classification. **This functionality is likely to changed in the near future.** Do not expect the current functionality to be supported by the Joy team.

2.3.10 num_pkts=N (number)

> Syntax:
>
> `num_pkts=N`

Limit the length of the `packets` array to at most `N`, where $0 \leq \mathtt{N} < 200$.

2.3.11 type=T (number)

> Syntax:
>
> `type=T`

Select message type: 1=SPLT, 2=SALT.

2.3.12 idp=N (number)

> Syntax:
>
> `idp=N`

The option `idp=N` reports the initial data packet of each flow, by including up to `N` bytes of that packet as hexadecimal. Note that **N is not boolean**; if you set `idp=1`, you will get only the first byte of that packet, which is probably not what you want.

The Initial Data Packet (IDP) is defined as the entire first packet in a flow, including the IP header, whose Data field has a nonzero length. For a TCP, UDP, or ICMP flow, the Data fields of those protocols are used to determine the IDP. For other protocols, the IP Data field is used.

2.3.13 label=L:F (string)

> Syntax:
>
> `label=L:F`

Add the label `L` to addresses that match the subnets in file `F`. Here the character `:` serves as a delimiter between the strings `L` and `F`. The JSON element `label` appears in the flow object whenever an address in the flow matches the subnet. This facility can be used to label flows that contact known-bad servers, for instance,

```
label=malware:badips.net
```

will add the label `malware` to all flows where `sa` or `da` is contained in a subnet in the file `badips.net`. The format of the subnet file is described in Section **??**.

2.3.14 URLmodel=URL (string)

> Syntax:
>
> `URLmodel=URL`

This option specifies the URL to be used to retrieve classifier updates.

2.3.15 model=F1:F2 (string)

> Syntax:
>
> `model=F1:F2`

Change the classifier parameters to use the SPLT parameters from file F1 and SPLT+BD parameters in file F2. Here the character `:` serves as a delimiter between the strings `F1` and `F2`. **This functionality is likely to changed in the near future.** Do not expect the current functionality to be supported by the Joy team.

2.3.16 hd=1 (boolean)

> Syntax:
>
> `hd=1`

If `hd=1`, include a header description summary. **This functionality is likely to changed in the near future.** Do not expect the current functionality to be supported by the Joy team.

2.3.17 URLlabel=URL (string)

> Syntax:
>
> `URLlabel=URL`

This option specifies the full URL, including filename, to be used to retrieve the label updates.

2.3.18 wht=1 (boolean)

> Syntax:
>
> `wht=1`

Include the bytewise Walsh-Hadamard Transform of the Data fields of the payload. This feature can detect some types of periodicity in data. **This functionality is likely to changed in the near future.** Do not expect the current functionality to be supported by the Joy team.

2.3.19 dns=1 (boolean)

> Syntax:
>
> `dns=1`

If `dns=1`, report DNS response information in the `dns` object at the top level of the JSON flow object. This object is a JSON array containing an object for each DNS response observed in each flow. An annotated example output:

```
"dns": [
    {
      "rn": "ocsp.digicert.com",          // requested name
      "rr": [                             // resource record
        {
          "cname": "cs9.wac.phicdn.net", // canonical name
          "ttl": 28407                    // seconds to live
        },
        {
          "a": "72.21.91.29",             // host address
          "ttl": 1204                     // seconds to live
        }
      ],
      "rc": 0                             // response code
    },
    {
      "rn": "detectportal.firefox.com",
      "rr": [
```

```
        {
          "cname": "detectportal.firefox.com.edgesuite.net",
          "ttl": 33
        },
        {
          "cname": "a1089.d.akamai.net",
          "ttl": 435
        }
      ],
      "rc": 0
    }
  ]
```

2.3.20 ssh=1 (boolean)

> Syntax:
>
> `ssh=1`

If `ssh=1`, report SSH information in the `ssh` object at the top level of the JSON flow object.

2.3.21 tls=1 (boolean)

> Syntax:
>
> `tls=1`

If `tls=1`, report TLS session metadata in the `tls` object at the top level of the JSON flow object.

2.3.22 dhcp=1 (boolean)

> Syntax:
>
> `dhcp=1`

If `dhcp=1`, report DHCP session metadata in the `dhcp` object at the top level of the JSON flow object.

2.3.23 http=1 (boolean)

> Syntax:
>
> `http=1`

If `http=1`, report HTTP session metadata in the `http` object at the top level of the JSON flow object. An annotated example follows:

```
"http": [
   {
      "out": [                          // request sent by client
         {
            "method": "GET"             // method
         },
         {
```

```
            "uri": "/success.txt"      // uniform resource identifier
        },
        {
            "version": "HTTP/1.1"      // protocol version
        },
        {
            "Host": "detectportal.firefox.com"  // start of headers
        },
        {
            "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:58.0) ... "
        },
        {
            "Accept": "*/*"
        },
        {
            "Accept-Language": "en-US,en;q=0.5"
        },
        {
            "Accept-Encoding": "gzip, deflate"
        },
        {
            "Cache-Control": "no-cache"
        },
        {
            "Pragma": "no-cache"
        },
        {
            "Connection": "keep-alive"
        },
        {
            "body": ""                      // empty HTTP body
        }
        ],
    "in": [                                 // response from server
        {
            "version": "HTTP/1.1"           // protocol version
        },
        {
            "code": "200"                   // response code
        },
        {
            "reason": "OK"                  // reason
        },
        {
            "Content-Type": "text/plain"    // start of headers
        },
        {
            "Content-Length": "8"
        },
        {
            "Last-Modified": "Mon, 15 May 2017 18:04:40 GMT"
        },
        {
            "ETag": ".ae780585f49b94ce1444eb7d28906123 ... "
        },
```

```
            {
               "Accept-Ranges": "bytes"
            },
            {
               "Server": "AmazonS3"
            },
            {
               "X-Amz-Cf-Id": "uYAJAK6Ts4c33EeScRKbMphDG ... "
            },
            {
               "Cache-Control": "no-cache, no-store, must-revalidate"
            },
            {
               "Date": "Tue, 23 Jan 2018 05:00:23 GMT"
            },
            {
               "Connection": "keep-alive"
            },
            {
               "body": "737563636573730a"    // (first bytes of) body
            }
         ]
      }
]
```

2.3.24 ike=1 (boolean)

> Syntax:
>
> `ike=1`

If `ike=1`, report IKE session metadata in the `ike` object at the top level of the JSON flow object.

2.3.25 payload=N (number)

> Syntax:
>
> `payload=N`

If `payload=1`, report the initial (up to) 32 bytes of the Data field of the initial data packet, as the hexadecimal string `payload` at the top level of the JSON flow object.

2.3.26 salt=1 (boolean)

> Syntax:
>
> `salt=1`

This functionality is likely to changed in the near future. Do not expect the current functionality to be supported by the Joy team.

2.3.27 ppi=1 (boolean)

Syntax:

```
ppi=1                          // include TCP per-packet info (ppi)
```

If ppi=1, report TCP per-packet information in the array ppi at the top level of the JSON flow object. An annotated example output:

```
"ppi": [
   {                                  // initial (SYN) packet
      "b": 0,                         // number of bytes in data field
      "seq": 1382708406,              // initial sequence number
      "ack": 0,                       // no acknowledgement number
      "rseq": 0,                      // relative sequence number
      "flags": "S",                   // flags: SYN
      "t": 0,                         // time since start of flow = 0ms
      "olen": 20,                     // option length = 20 bytes
      "opts": [                       // options
         {
            "mss": 1460               // maximum segment size
         },
         {
            "sackp": null             // selective acknowledgement OK
         },
         {
            "ts": {                   // timestamp
               "ecr": 0,              // echo reply = 0
               "val": 426827068       // value
            }
         },
         {
            "noop": null              // no-op (padding)
         },
         {
            "ws": 7                   // window scale = 7
         }
      ],
      "rack": 0,                      // relative ack number = 0
      "dir": ">"                      // direction: sa->da
   },
   {                                  // second packet; SYN/ACK
      "b": 0,                         // number of bytes in data field
      "seq": 712584705,               // initial sequence number (for responder)
      "ack": 1382708407,              // acknowledgement number
      "rseq": 0,                      // relative sequence number
      "flags": "SA",                  // flags: SYN, ACK
      "t": 38,                        // time since start of flow = 38ms
      "olen": 4,                      // option length = 4 bytes
      "opts": [                       // options
         {
            "mss": 1460               // maximum segment size
         }
      ],
      "rack": 1,                      // relative acknowledgement number
      "dir": "<"                      // direction: sa<-da
```

```
    },
    {                                   // third packet; ACK of responder's SYN
       "b": 0,                          // number of bytes in data field
       "seq": 1382708407,               // sequence number
       "ack": 712584706,                // acknowledgement number
       "rseq": 1,                       // relative sequence number
       "flags": "A",                    // flags: ACK
       "t": 38,                         // time since start of flow = 38ms
       "olen": 0,                       // option length = 0 bytes
       "opts": [],                      // options (none present)
       "rack": 1,                       // relative acknowledgement number
       "dir": ">"                       // direction: sa->da
    },
    {
       "b": 174,                        // bytes in data field
       "seq": 1382708407,
       "ack": 712584706,
       "rseq": 0,
       "flags": "PA",                   // flags: ACK, PSH
       "t": 38,
       "olen": 0,
       "opts": [],
       "rack": 1,
       "dir": ">"
    },
    ...
]
```

Chapter 3

The `sleuth` tool

3.1 Model

The `sleuth` program can be used to process the flow objects from JSON files or PCAP files; it can read from files or from `stdin`, filter out flows that match particular criteria, select data elements from flows, perform other types of analysis, and print out the results. Each flow is processed in order that it is read, by performing one or more operations on it. The arguments provided to the program determine the operations that are performed, and their ordering. By chaining together multiple operations, many different results can be computed.

3.2 Commands

The operations are roughly modeled the Structured Query Language (SQL) syntax.

select determines which elements will be selected from the flow; other elements are omitted

where sets a condition which the flows must match; all non-matching flows are omitted

dist computes a distribution (count and total) of the flows

groupby causes all flows with the same groupby value to be processed together

sum computes the sum over selected elements

3.2.1 Pseudo-JSON expressions

The `--where`, `--select`, and `--groupby` commands use pseudo-JSON notation to specify JSON objects of interest. A pseudo-JSON expression is formed from a JSON expression by removing the quotes around each name, leaving each name but removing each value, and removing colons. For instance, the pseudo-JSON expression

```
sp,dp,tcp{out{opts[{mss}]}}
```

matches the JSON object

```
{"sp":65150, "dp":443, "tcp":{"out":{"opts":[{"mss":1460}]}}}
```

In a manner of speaking, a pseudo-JSON object looks like the JSON objects it matches, with the values stripped out. Pseudo-JSON command line arguments should be enclosed in quotes, so that any commas that appear in those arguments do not confuse the shell. That is, the argument `--select "sa,da,sp,dp,pr"` should work with your shell, whereas `--select sa,da,sp,dp,pr` may not.

3.2.2 –pretty

The `--pretty` command causes the JSON output to be pretty-printed, in which case there is a single JSON element on each line, with indentation to promote readability.

3.2.3 –select

The `--select` command selects particular elements to be included in the objects; all other elements are excluded. The nesting structure of the objects is unchanged. The syntax is: `--select elementlist`

```
elementlist = element
            | elementlist "," pseudo-JSON-expression;
```

Each `element` is an element of the flow. The command `--select pr,dp` selects the protocol (pr) and destination port (dp), for instance, so that only those elements are included in the flows output by the command.

The command `--select packets` selects only the `packets` element, which is an array of objects containing the bytes, directions, and inter-packet arrival times:

```
sleuth --select packets

{'packets': [{'b': 45, 'ipt': 0, 'dir': '>'}, ... ]}
{'packets': [{'b': 380, 'ipt': 0, 'dir': '>'}, ... ]}
...
```

It is also possible to select particular elements within an object in an array, by using a JSON-like syntax to describe the selected elements. For instance, the following example selects only the bytes from the packets:

```
    sleuth --select packets[{b}]

{'packets': [{'b': 45 }, {'b': 261 }]}
{'packets': [{'b': 45 }, {'b': 261 }]}
{'packets': [{'b': 45 }, {'b': 261 }]}
{'packets': [{'b': 45 }, {'b': 287 }]}
```

```
sleuth --select "http[{out{User-Agent}]}"

{"http": {"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X  ... "}}
{"http": {"User-Agent": "AppleCoreMedia/1.0.0.15G1004 (Macintosh ... "}}
{"http": {"User-Agent": "ocspd/1.0.3"} }
```

3.2.4 –where

The `--where` command filters objects against a condition, so that the condition is true for each object output. The order of the objects is unchanged. The syntax is: `--where condition`

```
condition = predicate
          | "(" predicate ")"
          | predicate "|" predicate    (* or (disjunction)   *)
          | predicate "," predicate;   (* and (conjunction) *)

predicate = pseudo-JSON-expresson operator value;
```

```
operator  = "="              (* equals                      *)
          | "~"              (* does not equal              *)
          | ">"              (* greater than (numbers only) *)
          | "<";             (* less than (numbers only)    *)

value     = characterstring | number;
```

The condition `element=*` is true whenever `element` is present in an object, and `element~*` is true whenever `element` is not present in an object. The 'wildcard' values `*` and `?` can be included in character strings (one or more times), in which case `*` matches any sequence of characters and `?` matches any single character. If `element` includes an array, then its predicate is true if it holds for any value in the array. For instance, `--where ẗtp[{in[{Content-Type}]}]=*xml¨` matches any HTTP Content-Type that ends in "xml". This is also true if the element is an array of numbers or strings, as opposed to an array of objects, as shown in the following example:

```
sleuth --select tls{cs} --where "tls{cs[]}=c02b"

{"tls": {"cs": ["c02b", "c02f", "c00a", "c009", "c013", "c014", ... ]}}
{"tls": {"cs": ["1301", "1303", "1302", "c02b", "c02f", "cca9", ... ]}}
```

Each condition consists of a JSON field name, an operator, and value, such as `dp=80` (to process only HTTP traffic) or `ob>99` (to process only flows with 100 or more outbound bytes).

The supported operators are `>`, `<`, and `=`. If the field does not appear in a flow, then it does not match the condition. Quotation marks around the command argument containing the `>` and `<` symbols are needed in order to prevent them from being misinterpreted by the shell. The expression `field=*` will match any value of `field`.

`*` matches any string of characters

`?` matches any single character

`[seq]` matches any character in seq

For a literal match, surround the special characters in brackets; `[?]` matches the character `?`, for instance.

Elements contained in nested objects, or objects within arrays, can be included in a condition. The syntax is similar to that for the `select` operation**??**. For instance, `--where ''packets[b>900]''` selects flows that contain packets with `b` greater than 900. As above, the command syntax is suggestive of the form of the output.

For instance, `--where "da=10.*|da=172.16.*|da=192.168.*"` will select flows whose destination address is in the RFC 1918 [**?**] private address range:

```
sleuth --select da --where "da=10.*|da=172.16.*|da=192.168.*" --dist

{"count": 257, "total": 367, "da": "10.41.35.255"}
{"count": 99, "total": 367, "da": "10.32.222.255"}
{"count": 10, "total": 367, "da": "10.41.32.146"}
{"count": 1, "total": 367, "da": "10.41.32.238"}
```

The example above makes use of the `--dist` command (Section **??**).

3.2.5 –dist

The command `--dist` computes the distribution of the objects, and counts the number of times that each object appears in the stream, as well as the total number of objects; elements representing those

sums are included in the JSON objects as `count` and `total`, respectively. The objects are output in an order such that the `count` is decreasing. The `--dist` command does not have any arguments.

The `--dist` command is currently a memory hog, and it might fail on very large data inputs.

3.2.6 –groupby

The `--groupby` command splits the input stream of objects into one or more output streams, grouped by one or more elements. The command takes an element (such as `da`, or destination address) as a parameter, and then creates a pipeline for each distinct value of that object, and shunts each flow to the pipeline matching the value of its object. The syntax is:

```
--groupby elementlist

elementlist = element
            | elementlist "," pseudo-JSON-expresson;
```

For each `element` in `elementlist`, a separate output stream is created. The downstream processing is performed separately on each of those output streams. For instance:

`--groupby da` creates a separate stream for each destination address

`--groupby da,dp` creates a stream for each distinct (destination address, destination port) tuple

The `object` can be any element, including JSON-like objects (such as `pr,dp`). For instance, selecting on the start time (`ts`) and splitting on destination address (`da`) shows the result of the command:

```
sleuth --select da,ts --groupby da

{'ts': 1452263350.138106, 'da': u'87.250.250.119'}
{'ts': 1452263350.586721, 'da': u'87.250.250.119'}
{'ts': 1452263345.211611, 'da': u'173.38.117.77'}
{'ts': 1452263349.635237, 'da': u'185.84.108.11'}
{'ts': 1452263349.893659, 'da': u'185.84.108.11'}
{'ts': 1452263349.89913, 'da': u'185.84.108.11'}
{'ts': 1452263349.899592, 'da': u'185.84.108.11'}
{'ts': 1452263350.010724, 'da': u'185.84.108.11'}
{'ts': 1452263360.829828, 'da': u'216.58.219.238'}
{'ts': 1452263355.124145, 'da': u'161.44.124.122'}
{'ts': 1452263349.628012, 'da': u'64.102.6.247'}
{'ts': 1452263349.874641, 'da': u'64.102.6.247'}
{'ts': 1452263349.908992, 'da': u'64.102.6.247'}
{'ts': 1452263349.973846, 'da': u'64.102.6.247'}
```

The `--groupby` command is currently a memory hog, and it might fail on very large data inputs.

3.2.7 –sum

The command `--sum` computes the sum, across all objects, of one or more elements. The order of the objects may be changed. The syntax is:

```
--sum elementlist

elementlist = element
            | elementlist "," pseudo-JSON-expresson;
```

For each `element` in the `elementlist`, the value of that element from each object in the stream is added into a tally, which is written into an output object as the value of that element. The `elementlist` must contain at least one element, and may contain multiple elements, separated by commas. The other elements in the object *should* be constant.

3.2.8 –no_stitch

Normally, sleuth stitches together successive flow objects with the same flow key. The `--no_stitch` command prevents this stitching from taking place. This option should not normally be used; otherwise, a single long-lived flow might appear to be a succession of flows, and this situation may create ambiguity about flow direction.

3.2.9 –fingerprint

The command `--fingerprint fptypes` computes the fingerprints for each of the `fptypes` specified. elements. The syntax is:

```
--fingerprint fptypelist

fptypelist = fptypelist "," fptype | fptype;

    fptype = { "tls", "http", "tcp", "all" };
```

A `fingerprint` object is created at the top level of the flow object, and for each `fptype`, a fingerprint is computed and included in that object. Additionally, an `inferences` object is created at the top level of the flow object, which holds tls inferences about the fingerprints. The following examples show the structure of the `fingerprint` object, and the `fingerprint` and `inferences` objects for TLS:

```
sleuth *.pcap --fingerprint all --select dp,fingerprint

{"dp": 443, "fingerprint": {"tls": { ... }, "tcp": { ... }}
{"dp": 80,  "fingerprint": {"http": [ ... ], "tcp": { ... }}
```

```
sleuth *.pcap --fingerprint tls --select fingerprint,inferences

  {
    "inferences": {
      "tls": ["firefox-24.0"]
    },
    "fingerprint": {
      "tls": {
        "cs": [
          "c00a", "c009", "c013", "c014", "c008", "c012", "c007", "c011",
          "0033", "0032", "0045", "0044", "0039", "0038", "0088", "0087",
          "0016", "0013", "c004", "c00e", "c005", "c00f", "c003", "c00d",
          "c002", "c00c", "002f", "0041", "0035", "0084", "0096", "feff",
          "000a", "0005", "0004"
        ],
        "c_extensions": [
          {"server_name": null},
          {"renegotiation_info": "00"},
          {"supported_groups": "0006001700180019"},
          {"ec_point_formats": "0100"},
          {"session_ticket": null},
          {"kind": 13172, "data": null}]
      }
    }
  }
```

Joy keeps the `fingerprint` and `inferences` separate because the former is a simple deterministic function of the flow object, and the latter brings in auxiliary information and may change depending on what auxiliary information is available. More formally, a *fingerprint* is set of data features that are characteristic of the source, rather than the session. In `sleuth`, a fingerprint is constructed by selecting certain data elements from a flow object, and then normalizing some of them by setting their data elements to `null`. The normalization process removes session-specific information, while preserving information about the presence of data features. A good illustration of this process is given by a TCP fingerprint:

```
sleuth *.pcap --fingerprint tcp --select fingerprint --pretty

{
  "fingerprint": {
    "tcp": {
      "out": {
        "opt_len": 20,
        "opts": [
          { "mss": 1460 },
          { "sackp": null },
          { "ts": null },
          { "noop": null },
          { "ws": 7 }
        ]
      }
    }
  }
}
```

This fingerprint records the TCP options that are present as well as their order in the option list, and retains the JSON values that are characteristic of the client rather than the session (`mss`, `ws`), while eliminating session-specific data (such as the timestamp data in `ts`).

Currently, only TLS inferences are supported. The file `res_tls_fingerprints.json` in the `sleuth` package holds a set of fingerprints, and has the following format. Each line is a JSON object that contains the elements `label` and `fingerprint`. The former is a list of strings, and the latter is a fingerprint object as defined above. For example:

```
{"label": [ "firefox-31.0" ], "fingerprint": {"tls": { ... } } }
{"label": [ "firefox-31.0" ], "fingerprint": {"tls": { ... } } }
{"label": [ "firefox-24.0" ], "fingerprint": {"tls": { ... } } }
...
```

Fingerprint objects compatible with this file format can be created with the `sleuth --fingerprint` command.

3.2.10 –tls_sec

The `--tls_sec` option performs an in-depth cryptographic assessment of the flow against a cryptographic security policy. Any policy failures are reported as "concerns." There are several related flags that allow for customization of the assessment. If none of the related flags are specified, the defaults will be used. The flags are as follows:

`--policy_file file` runs the cryptographic assessment using the specified policy file, which defaults to `res_tls_policy.json`. This file defines security levels and assigns a security level to the attributes that will be found in the flow.

`--failure_threshold threshold_value` tells the tool to only report concerns for items that are `threshold_value` or less. By default this is set to 2 *legacy*

`--unknowns ignore/report` tells the tool to ignore unknown attributes rather than reporting them as failure. By default, this is set to report.

`--compliance fips_140/other` checks the cryptographic configurations for compliance against the specified policy, which must be present in `res_tls_compliance.json`

The output will display in a `tls_sec` attribute, which enumerates the security level classification as well as the concerns based on the policy and compliance models (here we didn't explicitly specify files, so the defaults are used) as follows:

```
sleuth *.pcap --tls_sec --select "tls_sec,da,sa" --pretty

{
   "sa": "x.x.x.x",
   "tls_sec": {
      "seclevel": {
         "concerns": [
            "no certs data",
            "enc - 3DES_EDE_CBC",
            "hash - SHA"
         ],
         "classification": "legacy"
      }
   },
   "da": "x.x.x.x"
}
```

Chapter 4

Examples

Who uses TLS server certificate chains where one or more of the keys is smaller than 2048 bits?

```
sleuth *.pcap --select "tls{s_cert[{signature_key_size,signature_algo}]},da" \
              --where "tls{s_cert[{signature_key_size}]}<2048" --pretty

{
   "tls": {
      "s_cert": [
         {
            "signature_key_size": 2048,
            "signature_algo": "sha256WithRSAEncryption"
         },
         {
            "signature_key_size": 2048,
            "signature_algo": "sha256WithRSAEncryption"
         },
         {
            "signature_key_size": 1024,
            "signature_algo": "sha1WithRSAEncryption"
         }
      ]
   },
   "da": "23.196.114.245"
}
{
   "tls": {
      "s_cert": [
         {
            "signature_key_size": 2048,
            "signature_algo": "sha256WithRSAEncryption"
         },
         {
            "signature_key_size": 2048,
            "signature_algo": "sha256WithRSAEncryption"
         },
         {
            "signature_key_size": 1024,
            "signature_algo": "sha1WithRSAEncryption"
         }
```

```
      ]
   },
   "da": "72.163.4.161"
}
```

```
# show flows that contain a packet whose payload is exactly 50 bytes in length
#
./sleuth *.pcap --where "packets[{b}]=50" --select "da,packets[{b}]"

{
  "packets": [
    {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68},
    {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68}, {"b": 68},
    {"b": 50}, {"b": 68}, {"b": 50}, {"b": 68}, {"b": 50}, {"b": 68},
    {"b": 68}, {"b": 50}, {"b": 50}, {"b": 50}
  ],
  "da": "172.16.255.255"
}
```

Chapter 5

JSON Schema

The JSON output from Joy is, by default, compressed with the `GZIP` algorithm. In its compressed form, it can be conveniently read with the `zless` program or piped to the standard output with `zcat`. In it uncompressed form, there is a single JSON object per line of the file. A `metadata` object always contains the `version` field, which indicates the version of the `joy` capture tool used to create it. This object contains the complete configuration of that program, which can be important to know when analyzing the flow objects that it produced. When `joy` is run in online mode, it writes its output to a file or series of files, and the first object that it writes is a metadata object describing its configuration.

Flow objects comprise most of the output. The capture tool writes out flow objects in the order that the flows were created; the `time_start` fields are in increasing (or at least non-decreasing) order. The flow objects will contain different data elements, depending on what protocols the flow contained.

5.1 JSON model for network data

JavaScript Object Notation (JSON [**?**]) excels at representing complex data while at the same time being readable and easily extensible. That readability is essential making the data useful; a person who does not understand the JSON format cannot write queries or programs against it. This note explains the conventions that the Joy package uses to represent network data. Its goals are to faithfully represent as much data as possible, including details such as the ordering of data elements and the capitalization of text strings, and to have a schema that contains all of the important data elements as explicitly named members. This note explains the conventions used to meet these goals.

5.1.1 TLVs

Many network protocols make use of a list or array of elements of different types, with each element containing a type code that indicates how it should be processed. In the Type/Length/Value (TLV) data representation, a message contains a list of elements, each of which consists of a type code field containing an integer that indicates how the element should be interpreted, a length field that indicates the number of bytes in the element, and a value field that contains the actual data. The flexibility of the TLV representation makes it well suited for network protocols, and its use is commonplace. TCP options [**?**, **?**] are an early example, though the acronym TLV appeared later and was not used in the TCP specification; we use that example to illustrate the Joy data-representation conventions. Table **??** summarizes the important TCP option types.

In a TLV scheme, some type codes are *registered*, that is, the format and interpretation of the value field for that type is well defined and documented. The data formats of known types are typically defined in standards documents, and the type codes for IETF standards are registered with the Internet Assigned Number Authority (IANA) [**?**], as is the case for TCP Options [**?**]. Each known type type has a (human readable) name, such as Maximum Segment Size (MSS), Window Scale (WS), or Time Stamp (TS). A type code that is not registered may be *unassigned* or *reserved*. An unassigned type code is one that does not correspond to any registered data format, and thus it is unknown how to interpret its

Figure 5.1: An illustration of a bidirectional TCP flow.

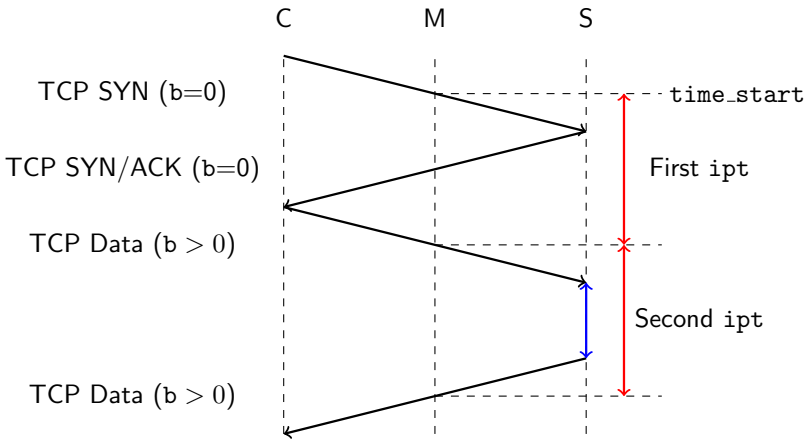


Table 5.1: The well-known TCP options.

Type	Meaning	Joy Name	Joy Value
0	End of Option List	-	-
1	No-Operation	`noop`	`none`
2	Maximum Segment Size	`mss`	number
3	Window Scale	`ws`	number
4	SACK Permitted	`sackp`	`none`
5	SACK	`sack`	hex string
8	Timestamps	`ts`	object

value field. Sometimes implementers define a new option but neglect to register it with IANA, in which case IANA will typically reserve that type code, so that it is moved out of the unassigned category. We say that a type code and its corresponding data format are *known* whenever they are registered, or whenever they are reserved but with a known format.

5.1.2 Conventions

When representing a list of TLV elements in JSON, Joy uses the following conventions. The list is represented in JSON as an array of objects, ordered as the elements appear in the message, and each object as follows:

- if the type code is known, the object contains a member whose JSON name is the human-readable name associated with the type code, such as `mss` for the TCP MSS option, and whose JSON value can be a string, array, `null`, `false`, `true`, or object, as is appropriate to represent the value of that particular TLV element,

- if the type is unknown, the object contains a member with the JSON name `type` whose JSON value is a numeric representation of the type code, and a member whose JSON name is `data` whose value is a hexadecimal representation of the value from the TLV element.

Additionally, all member names contain only non-whitespace characters, so that those names need not be quoted in JSON tools.

With these conventions, there is no need for a user or a program to memorize or understand the type codes that have been registered. Known types are represented as recognizable member names in the JSON schema, and unknown types are represented by `type` and `data` fields. Known types can easily be accessed, while the schema for unknown types is regular. Importantly, the conventions ensure that the JSON schema contains the information needed to understand the JSON data. The important

information is present in JSON member names, instead of residing in a JSON value. For instance, the TCP options are represented as

```
{
   "opts": [
      {
         "mss": 1460
      },
      {
         "sackp": null
      },
      {
         "ts": {
            "val": 4332608,
            "ecr": 0
          }
      },
      {
         "noop": null
      },
      {
         "ws": 7
      },
      {
         "type": 79,
         "data": "abadcafe"
      }
   ]
}
```

The JSON schema for TCP options is:

```
{
  "required": [
    "opts"
  ],
  "type": "object",
  "properties": {
    "opts": {
      "items": {
        "type": "object",
        "properties": {
          "sackp": {
            "type": "null"
          },
          "mss": {
            "type": "integer"
          },
          "ts": {
            "required": [
              "ecr",
              "val"
            ],
            "type": "object",
            "properties": {
              "ecr": {
                "type": "integer"
              },
              "val": {
                "type": "integer"
              }
            }
          },
          "noop": {
            "type": "null"
          },
          "ws": {
            "type": "integer"
          },
          "data": {
            "type": "string"
          }
        }
      },
      "type": "array"
    }
  }
}
```

What does it look like when these conventions aren't followed?

In contrast, the method of having the JSON schema for a TLV element contain members with the JSON names type, length, and data makes that schema devoid of the detailed information of how to interpret the data values, and forces the JSON user to use the `type` member as an index or search value for the `data`, and forces the JSON tools to interpret the `data` member in a type-specific way. This method results in a schema of

```
"opts": {
  "properties": {
```

```
        "type": "number",
        "length": "number",
        "data": "string"
    }
}
```

which gives no information about what type numbers might appear in the JSON. Furthermore, it is awkward to search with this schema, since searching for the `data` JSON values corresponding to particular `type` JSON value requires that the search tool report one value based on a search query corresponding to a different value.