Joy Library API

User's Guide

Overview

Joy is a BSD-licensed software package for extracting data features from live network traffic or packet capture (pcap) files, using a flow-oriented model similar to that of IPFIX or Netflow, and then representing these data features in JSON. It also contains analysis tools that can be applied to these data files. Joy can be used to explore data at scale, especially security and threat-relevant data.

Joy is intended for use in 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.

Relation to Cisco ETA

Joy has helped support the research that paved the way for Cisco's Encrypted Traffic Analytics (ETA), but it is not directly integrated into any of the Cisco products or services that implement ETA. The classifiers in Joy were trained on a small dataset several years ago, and do not represent the classification methods or performance of ETA. The intent of this feature is to allow network researchers to quickly train and deploy their own classifiers on a subset of the data features that Joy produces. For more information on training your own classifier, see analysis/README or reach out to joy-users@cisco.com.

Why an API and Library?

Joy was originally written as a research project. Once the value of Joy was established and products began incorporating Joy concepts, it became necessary to modify the Joy code so that products could import the code directly instead of re-writing the same ideas in a different form. The most prudent way to accomplish this goal is to turn the Joy code into a library with a well-defined API that products can link against. This promotes code re-use and single implementations of algorithms and concepts.

Key Concepts

LIBPCAP

Joy is a packet processing tool that is heavily modeled after libpcap. In fact, the libpcap constructs are used as the basis for the main entry into the API. Libpcap offers a proven and widely used format for packet processing, so we did not want to re-invent the wheel around these aspects of network packet processing.

JSON

Since Joy is a network analysis tool, there was a need for the output to be in a modern consumable format. The Joy team chose JSON as the output vehicle. JSON is readily consumable by most programming languages or modern tools. The Joy library offers an API to produce the JSON output if that is what you desire.

Anonymization

Since Joy deals with raw packet data, and Joy can produce output that includes things like IP addresses and usernames, it became necessary to offer the ability to anonymize any personally identifying data. The Joy code was originally written to offer this capability and the API into the library provides this ability as well.

IPFix Export

Joy has the ability to export its flow data using IPFix with Encrypted Traffic Analytics (ETA) data items. The Library API has specific configuration items around IPFix which allow the data to be transmitted to the desired IPFix collector. These configuration items are detailed later in the document.

High Level Architecture

This guide focuses on the Joy library and the API used to interface with the library. The architecture is similar to any library a developer might use. The Joy library does contain a few concepts which make the order of certain API calls important for correctly functioning software. The picture below outlines the overall flow of a sample program and its interactions with the Joy library.

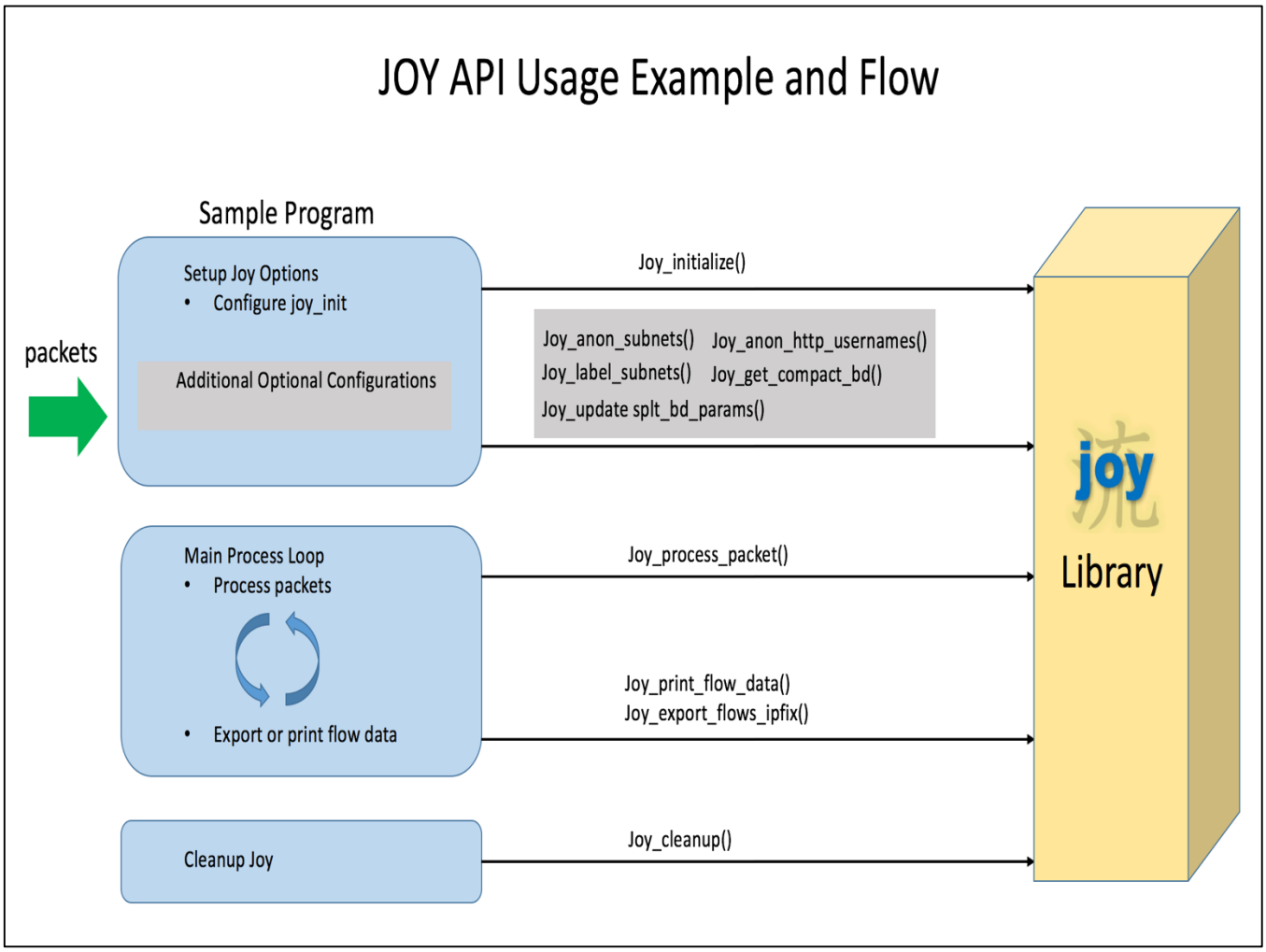

Figure 1. Single Context Processing Flow

As you can see in the usage picture, there are essentially 3 phases of using the Joy library. The first phase revolves around initializing the library to process and behave as you would like it to on your data. The second phase actually handles the processing and periodic output of the analyzed flow data. The output can be either simple printing of the flow data to JSON or actually exporting the flow data to another entity over IPFix. The final phase is just cleaning up once we are done processing packets.

Some applications may want to divide up packet processing to achieve better performance numbers or just segregate different subnets from one another. The following diagram shows how run the Joy library with multiple threads and contexts.

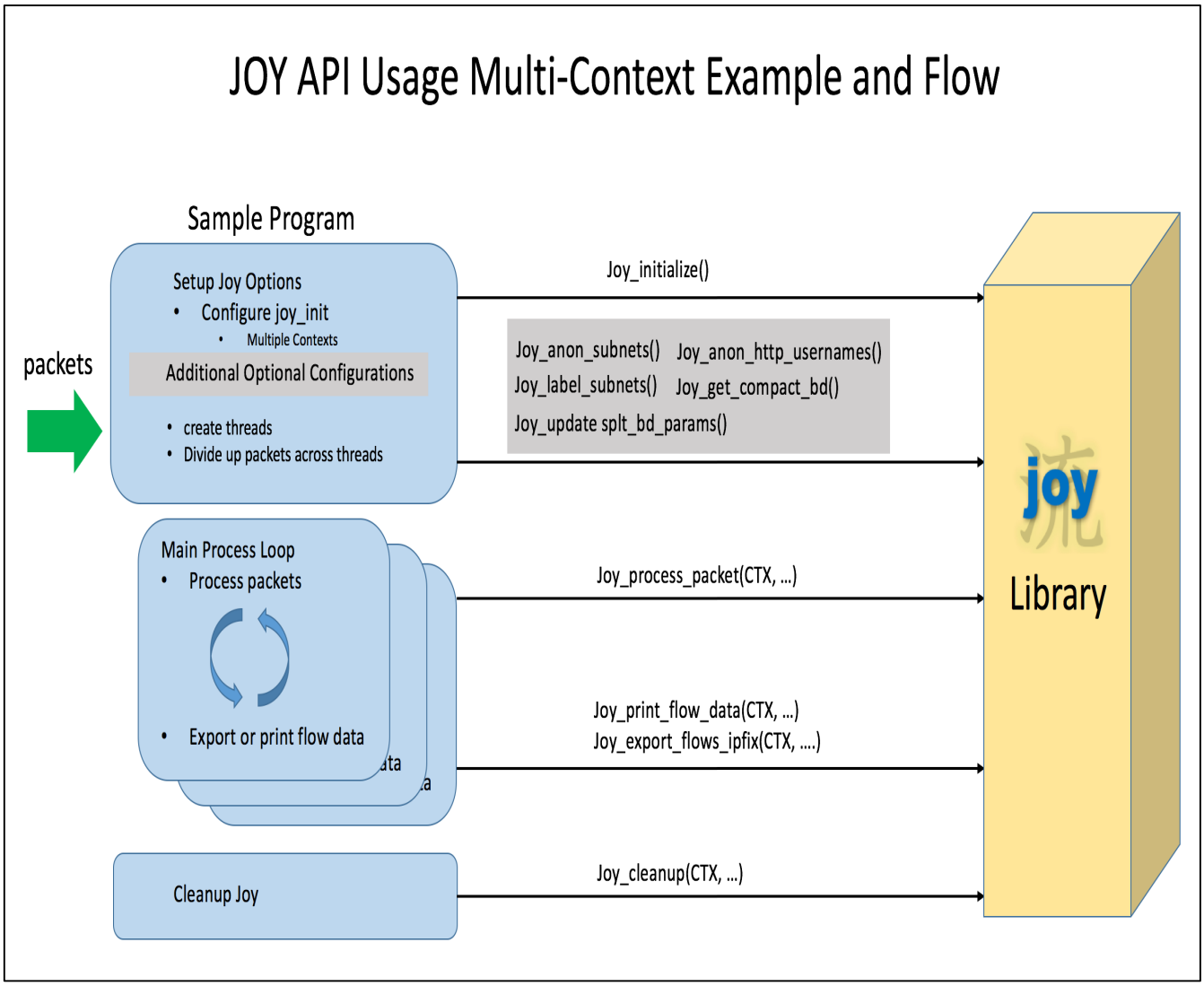

Figure 2. Multi-Context Processing Flow

As you can see in the usage picture, the 3 phases of using the Joy library are still present. What is different is the application/caller divides up the packets using a scheme appropriate for its purposes and sends the data to the correct context.

Initialization APIs

This section of the document describes the various initialization APIs that are available in the Joy library.

joy_initialize

**** This must be the first API called when using the Joy library ****

```
int joy_initialize (struct joy_init *data, char *output_dir, char
*output_file, char *logfile);
```

```
/*
 * Function: joy_initialize
 *
 * Description: This function initializes the Joy library
 *      to analyze the data features defined in the bitmask.
 *      If the IPFIX_EXPORT option is turned on, we will set
 *      additional items related to the export. The caller
 *      has the option to change the output destinations.
 *
 *      joy_initialize must be called before using any of the other
 *      API functions.
 *
 * Parameters:
 *      init_data - structure of Joy options
 *      output_dir - the destination output directory
 *      output_file - the destination outputfile name
 *      logfile - the destination file for errors/info/debug messages
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

The **joy_init** structure contains additional initialization items that are useful for configuring behavior and IPFix export capabilities.

```
/* structure used to initialize joy through the API Library */
struct joy_init {
    int type;            /* type 1 (SPLT) 2 (SALT) */
    int verbosity;       /* verbosity 0 (off) - 5 (critical) */
    int idp;             /* idp size to report, recommend 1300 */
    char *ipfix_host;    /* ip string of the host to send IPFix data to */
    uint32_t ipfix_port; /* port to send IPFix to remote on */
    uint32_t bitmask;    /* bitmask representing which features are on */
};
```

joy_init => type

The **type** parameter in the structure determines whether Sequence of Packet Lengths and Times (SPLT) or Sequence of Application Lengths and Times (SALT) is used during processing. The default is SPLT.

joy_init => verbosity

The **verbosity** parameter in the structure determines the amount of debug messages you see from the Joy Library. A value of 4 is recommended for most use cases of the Joy library. This means with a verbosity level of 4, you will see error and critical messages in the logfile. The various verbosity levels are defined below.

```
0   -   Off (no debug messages)
1   -   Debug (debug level messages and above are display)
2   -   Info (info level messages and above are displayed)
3   -   Warning (warning level messages and above are displayed)
4   -   Error (error level messages and above are displayed)
5   -   Critical (critical level messages and above are displayed)
```

joy_init => idp

The **idp** parameter in the structure determines how many bytes of the initial data packet (idp) are produced in the resulting output. It is recommended to use a value of 1300.

joy_init => ipfix_host

The **ipfix_host** parameter in the structure determines where the IPFix packets will be sent when in IPFix exporter mode. This parameter is only used when in IPFix exporter mode. To enable IPFix exporter mode, the joy_init.bitmask must have JOY_IPFIX_EXPORT_ON in it.

joy_init => ipfix_port

The **ipfix_port** parameter in the structure determines what port the IPFix packets will be sent on locally and what port the IPFix packets will be received on remotely when in IPFix exporter mode. This parameter is only used when in IPFix exporter mode. To enable IPFix exporter mode, the joy_init.bitmask must have JOY_IPFIX_EXPORT_ON in it.

joy_init => bitmask

The **bitmask** parameter in the structure determines what data features are enabled in the Joy library.

```
/*
 * Joy Library Bitmask Values
 *
 *    Bitmask values for turning on various network data features.
 *    Each value represents a feature within the Joy Library and
 *    whether or not it is turned on.
 *
```

```
 */
#define JOY_BIDIR_ON           0x01
#define JOY_DNS_ON             0x02
#define JOY_SSH_ON             0x04
#define JOY_TLS_ON             0x08
#define JOY_DHCP_ON            0x10
#define JOY_HTTP_ON            0x20
#define JOY_IKE_ON             0x40
#define JOY_PAYLOAD_ON         0x080
#define JOY_EXE_ON             0x100
#define JOY_ZERO_ON            0x200
#define JOY_RETRANS_ON         0x400
#define JOY_BYTE_DIST_ON       0x800
#define JOY_ENTROPY_ON         0x1000
#define JOY_CLASSIFY_ON        0x2000
#define JOY_HEADER_ON          0x4000
#define JOY_PREMPTIVE_TMO_ON   0x8000
#define JOY_IPFIX_EXPORT_ON    0x10000
```

Multiple options can be enable simply by ‘ORing’ the value together as such:

```
Joy_init.bitmask = (JOY_BIDIR_ON | JOY_TLS_ON | JOY_HTTP_ON);
```

output_dir

The **output_dir** parameter specifies where you want output directed. If this is NULL, then the output_dir will be set to the current directory (‘./’).

output_file

The **output_file** parameter specifies what file you want output directed. If this is NULL, then the output_file will be set to ‘stdout’.

logfile

The **logfile** parameter specifies what file you want logging information directed. If this is NULL, then the logfile will be set to ‘stderr’.

joy_print_config

```
int joy_print_config (int format);
```

```
/*
 * Function: joy_print_config
 *
 * Description: This function prints out the configuration
 *      of the Joy library in either JSON or terminal format.
 *
 * Parameters:
 *      format - JOY_JSON_FORMAT or JOY_TERMINAL_FORMAT
 *
 * Returns:
 *      none
 *
 */
```

format

The **format** parameter determines how the Joy library configuration is printed. There are two formats; JSON and standard line by line terminal output.

joy_anon_subnets

```
int joy_anon_subnets (char *anon_file);
```

```
/*
 * Function: joy_anon_subnets
 *
 * Description: This function processes a file of subnets to
 *      anonymized when processing the packet/flow data.
 *
 * Parameters:
 *      anon_file - file of subnets to anonymize
 *
 * Expected format of the file:
 *
 * # 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
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

anon_file

The **anon_file** parameter is a filename of subnets that requires the IP addresses to be anonymized. In this file, comments are denoted by a '#' character. It is expected on a single line, at most one subnet will be found. A single line may be nothing but comments or blank. Within a single line, comments and a subnet may be inter-mixed.

joy_anon_http_usernames

```
int joy_anon_http_usernames (char *anon_http_file);
```

```
/*
 * Function: joy_anon_http_usernames
 *
 * Description: This function processes a file of usernames
 *      to anonymized when processing the packet/flow http data.
 *
 * Parameters:
 *      anon_http_file - file of usernames to anonymize
 *
 * Expected format of the file:
 * username1
 * username2
 *     .
 *     .
 *     .
 * usernameN
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

anon_http_file

The **anon_http_file** parameter is a filename of user IDs to be anonymized. This file expects a single user ID per line and nothing else.

joy_update_splt_bd_params

```
extern int joy_update_splt_bd_params (char *splt_filename, char
*bd_filename);

/*
 * Function: joy_update_splt_bd_params
 *
 * Description: This function processes two files to update the
 *      values used for SPLT and BD processing in the machine learning
 *      classifier. The format of the file should match the format
 *      produced from the python program (model.py) from the
 *      Joy repository.
 *
 * Parameters:
 *      splt_filename - file of SPLT values
 *      bd_filename - file of BD values
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

splt_filename

The **splt_filename** parameter is a filename of SPLT (sequence of packet length and time) values to be used in the machine learning classifier. These values will replace the hardcoded default values that reside in the library. The format of this file must conform to the format the analysis/model.py produces.

bd_filename

The **splt_filename** parameter is a filename of BD (byte distribution) values to be used in the machine learning classifier. These values will replace the hardcoded default values that reside in the library. The format of this file must conform to the format the analysis/model.py produces.

joy_get_compact_bd

```
extern int joy_get_compact_bd (char *filename);
```

```
/*
 * Function: joy_get_compact_bd
 *
 * Description: This function processes a file to update the
 *      compact BD values used for counting the distribution
 *      in a given flow.
 *
 * Parameters:
 *      filename - file of compact BD values
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

filename

The **filename** parameter is a filename of compact BD (byte distribution) values to be used when counting the byte distribution in a flow.

joy_label_subnets

```
extern int joy_label_subnets (char *label, int type, char* filename);
```

```
/*
 * Function: joy_label_subnets
 *
 * Description: This function applies the label to the subnets specified
 *      in the subnet file.
 *
 * Parameters:
 *      label - label to be output for the subnets
 *      type - JOY_SINGLE_SUBNET or JOY_FILE_SUBNET
 *      subnet_str - a subnet address or a filename that contains subnets
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

label

The **label** parameter is a descriptive string to be associated with a subnet.

type

The **type** parameter determines if we are processing a single string that contains a subnet or if we are processing a file of subnets.

subnet_str

The **subnet_str** parameter is either a single string of a subnet and mask or a filename of subnets and masks to be labeled with the corresponding label. This API can be called multiple times for various labels and subnet files.

```
Sample contents of subnet_file.txt:
172.16.0.0/12
192.168.0.0/16

joy_label_subnets("myLabLabel", JOY_SINGLE_SUBNET, "10.0.0.0/8");
joy_label_subnets("myOtherLabel", JOY_FILE_SUBNET, "subnet_file.txt");
```

Processing APIs

This section of the document describes the various processing APIs that are available in the Joy library.

joy_process_packet

**** This API follows the libpcap prototype for handling packets ****

```
void joy_process_packet (unsigned char *ctx_index, const struct pcap_pkthdr
*header, const unsigned char *packet);
```

```
/*
 * Function: joy_process_packet
 *
 * Description: This function is formatted to match the libpcap
 *      prototype for processing packets. This is essentially
 *      wrapper function for the code used within the Joy library.
 *
 * Parameters:
 *      ctx_index - index of the context to use
 *      header - libpcap header which contains timestamp, cap length
 *               and length
 *      packet - the actual data packet
 *
 * Returns:
 *      none
 *
 */
```

ctx_index

The **ctx_index** parameter is signals which library context to use when processing this packet. If you are using one of the PCAP APIs to handle packet processing, specify the context to use in the *"user"* parameter. This will ensure that the packet gets to correct processing context. For example, to send the processing to the *"third"* context using *pcap_dispatch*, you would do the following:

```
index = 2; /* contexts are numbered 0 to MAX_LIB_CONTEXTS */
more = pcap_dispatch(handle, NUM_PACKETS_IN_LOOP, joy_process_packet,
(unsigned char*)index);
```

This results in the joy_process_packet API being called with the following information:

```
joy_process_packet(unsigned char *ctx_index, const struct pcap_pkthdr
                   *header,const unsigned char *packet)
ctx_index = 2
header = PCAP Packet Header Information
packet = the actual data packet
```

If you are not using LIBPCAP or something similar to handle your packet processing and can directly call joy_process_packet, then you would simply provide the context index as the first parameter in the API as such:

```
joy_context_index = 2;
joy_process_packet(unsigned char *)joy_context_index, header, packet);
```

header

The **header** parameter is a libpcap structure. This parameter contains the timestamp and lengths of the packet that was captured. The structure of this parameter is as follows:

```
struct pcap_pkthdr {
        struct timeval ts;      /* time stamp */
        bpf_u_int32 caplen;     /* length of portion present */
        bpf_u_int32 len;        /* length this packet (off wire) */
};
```

packet

The **packet** parameter is a character pointer to the actual packet data that was captured for analysis.

joy_print_flow_data

```
void joy_print_flow_data (unsigned int index, int type);
```

```
/*
 * Function: joy_print_flow_data
 *
 * Description: This function is prints out the flow data from
 *      the Joy data structures to the output destination specified
 *      in the joy_initialize call. The output is formatted as
 *      Joy JSON objects.
 *      Part this operation will check to see if there is any
 *      host flow data to collect, if the option is turned on.
 *
 * Parameters:
 *      index - index of the context to use
 *      type - JOY_EXPIRED_FLOWS or JOY_PRINT_ALL_FLOWS
 *
 * Returns:
 *      none
 *
 */
```

index

The **index** parameter determines which library context to print the flow data from.

type

The **type** parameter determines whether or not all flows or just expired flows are printed. When this API is called, any flows that are printed are subsequently removed from the flow table.

joy_export_flows_ipfix

```
void joy_export_flows_ipfix (unsigned int index, int type);
```

```
/*
 * Function: joy_export_flows_ipfix
 *
 * Description: This function is exports the flow data from
 *      the Joy data structures to the destination specified
 *      in the joy_initialize call. The flow data is exported
 *      as IPFix packets to the destination.
 *
 * Parameters:
 *      index - index of the context to use
 *      type - JOY_EXPIRED_FLOWS or JOY_ALL_FLOWS
 *
 * Returns:
 *      none
 *
 */
```

index

The **index** parameter determines which library context to export the flow data from.

type

The **type** parameter determines whether or not all flows or just expired flows are exported over IPFix. When this API is called, any flows that are exported are subsequently removed from the flow table.

Cleanup APIs

This section of the document describes the various cleanup APIs that are available in the Joy library.

joy_cleanup

```
void joy_cleanup (unsigned int index);
```

```
/*
 * Function: joy_cleanup
 *
 * Description: This function cleans up any leftover data that maybe
 *      hanging around. If IPFix exporting is turned on, then it also
 *      flushes any remaining records out to the destination.
 *
 * Parameters:
 *      index - index of the context to use
 *
 * Returns:
 *      none
 *
 */
```

index

The **index** parameter determines which library context to clean up.

This API should only be called when you are finished sending packets the Joy library and do not wish to process packets anymore. This API will flush out any remaining data and then free up the memory structures for the flow records.