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 pictures below outline the overall flow of a sample program and its interactions with the Joy library.

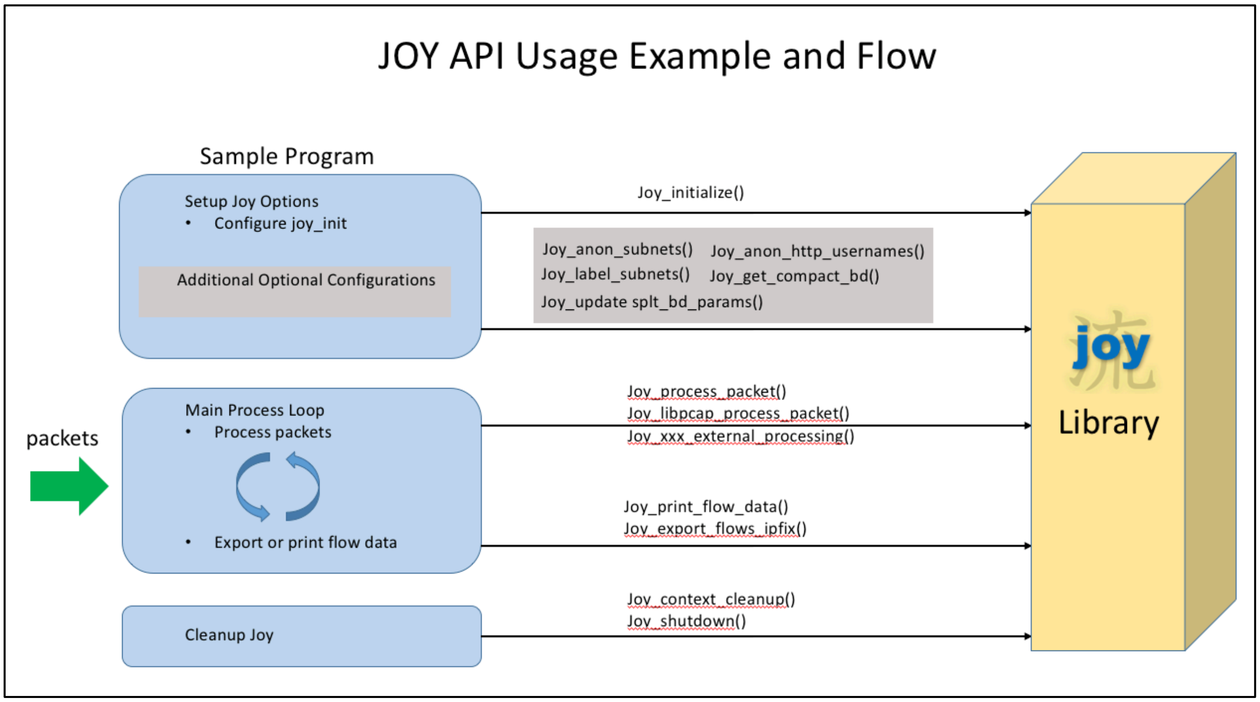
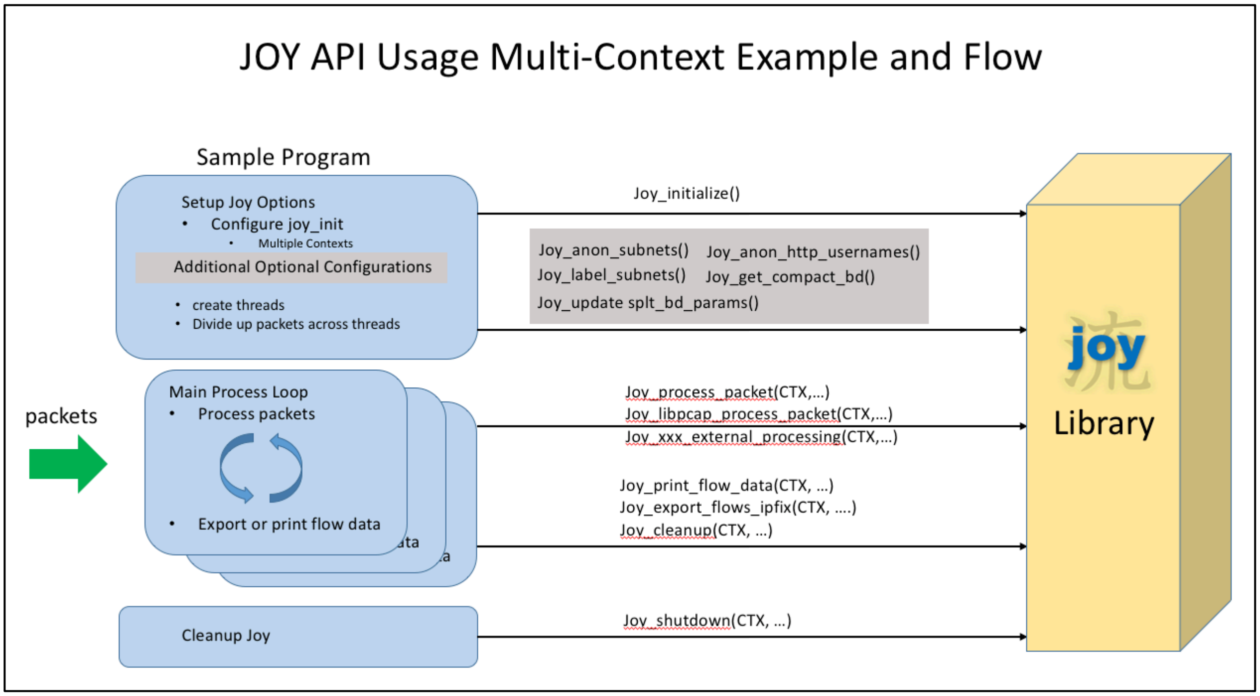


Figure 1. Single Context Processing Flow

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. In some cases the application may wish to handle the flow records itself. There are specific APIs for “external processing” in those cases. 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.



JOY API Usage Multi-Context Example and Flow

Sample Program

Setup Joy Options
- Configure joy_init
  - Multiple Contexts

Additional Optional Configurations
- create threads
- Divide up packets across threads

Joy_initialize()

Joy_anon_subnets() Joy_anon_http_usernames()
Joy_label_subnets() Joy_get_compact_bd()
Joy_update splt_bd_params()

packets

Main Process Loop
- Process packets
- Export or print flow data

Joy_process_packet(CTX,...)
Joy_libpcap_process_packet(CTX,...)
Joy_xxx_external_processing(CTX,...)

Joy_print_flow_data(CTX, ...)
Joy_export_flows_ipfix(CTX, ...)
Joy_cleanup(CTX, ...)

Cleanup Joy

Joy_shutdown(CTX, ...)

joy
Library

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.

For a multi-threaded and multi-context environment, the JOY library separates the context workers. This means that each context worker essentially has its own flow-record table and data analysis pieces. As such, in order to get accurate data analysis, you will want to keep the packets for a given flow going to the same context worker. Additionally, if you wish to see the bi-directional flow, then you will want to have the opposite direction packets going to the same context worker as well.

The JOY library provides a default traffic distribution mechanism which uses the 5-tuple from a packet to generate an appropriate context to use for processing. The default algorithm takes into account bi-directional flows and keep those packets grouped to the same context worker. The algorithm does a fairly good job of equal distribution across the number of worker contexts that were specified by the parent application at JOY initialization time.

At any given time, a single thread should be the **only** process operating on a given context. This means you could run into issues if multiple threads are modifying the same JOY Library context at the same time. This would imply that in a multi-context usage of the JOY library, you have a traffic distribution algorithms dividing up the work and keep packets within the same flow going to the same context workers.

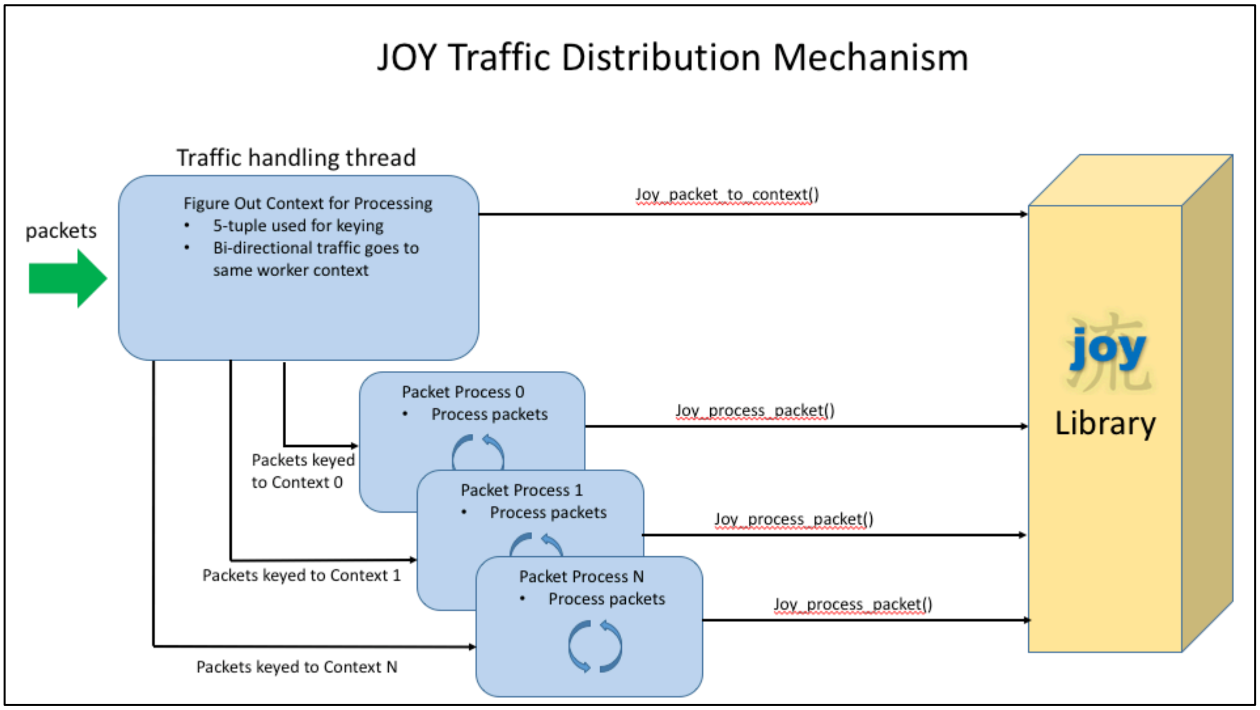

JOY Traffic Distribution Mechanism
Traffic handling thread
packets
Figure Out Context for Processing
• 5-tuple used for keying
• Bi-directional traffic goes to same worker context
Joy_packet_to_context()
Packet Process 0
• Process packets
Packets keyed to Context 0
Joy_process_packet()
Packet Process 1
• Process packets
Packets keyed to Context 1
Joy_process_packet()
Packet Process N
• Process packets
Packets keyed to Context N
Joy_process_packet()
joy
Library

Figure 3. Traffic Distribution

The picture above outlines the usage of the default JOY library traffic distribution algorithm. As you can see, there is one process whose job is to do nothing but get packets, figure out which context to use for processing them and then sending the packet to that context. The actual data processing and analysis is done in a separate thread for each context worker.

The bulk of the work is done in the specific worker context packet processing thread. The "packet handler" is really just a filter for the various worker threads. If an application had an optimized "packet filter" or a hardware mechanism for distributing packets to worker threads, they could achieve even high efficiencies.

Example Programs

In the github repository, there are two example programs for using the JOY APIs and Library. These programs are for demonstration purposes and provide a simplistic view on how to use the API and Library.

The first program is in **joy_api_test.c**. This program is a mutli-threaded program that will process pcap file in multiple JOY library contexts. This program demonstrates the division of global APIs for configuration and behavior versus the context specific APIs for processing packets ad generating output. The “main” of this program simply sets up the Joy Library with specified data features and behavior parameters, instantiates the worker threads, each worker thread then processes a pcap file, and finally the parent process performs the necessary cleanup. This program expects 3 pcap files to be identified on the command line so each of the 3 workers will have a file to process. The usage for this program is:

- ➢ bin/joy_api_test pcap1.pcap pcap2.pcap pcap3.pcap

The second program is in **joy_api_test2.c**. This program is a single threaded program that will process hardcoded packets and export them using the IPFix exporter in the Joy Library. The program initializes the JOY Library, sets up the data features and configuration parameters, initializes the IPFix exporter, processes the hardcoded packets, exports the flow records over IPFix and finally cleans up the JOY Library. The usage for this program is:

- ➢ bin/joy_api_test2

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 *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 */
typedef struct joy_init {
    uint8_t verbosity;         /* verbosity 0 (off) - 5 (critical) */
    uint32_t max_records;      /* max record in output file */
    uint16_t num_pkts;         /* num_pkts to report on per flow */
    uint8_t contexts;          /* number of contexts the app wants to use */
    uint16_t inact_timeout;    /* seconds for inact timeout - 0 = default used */
    uint16_t act_timeout;      /* seconds for act timeout - 0 = default used */
    uint16_t idp;              /* idp size to report, recommend 1300 */
    const char *ipfix_host;    /* ip string of the host to send IPFix data to */
    uint16_t ipfix_port;       /* port to send IPFix to remote on */
    uint32_t bitmask;          /* bitmask representing which features are on */
} joy_init_t;
```

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 => max_records

The **max_records** parameter in the structure determines how many flow records are written to the active output file before the file is rotated to a new file. This allows a user to specify the size of the output files that the JOY library will produce. The default is Zero, meaning all records will go into a single output file.

joy_init => num_pkts

The **num_pkts** parameter in the structure determines how many packets will be used for flow feature data. The default is 50 packets.

joy_init => contexts

The **contexts** parameter in the structure determines how many worker threads the application wishes to use when processing flow data. The default is 1 context meaning a single threaded application.

joy_init => inact_timeout

The **inact_timeout** parameter in the structure determines the number of seconds for the inactive timeout value. The default is 10 seconds.

joy_init => act_timeout

The **act_timeout** parameter in the structure determines the number of seconds for the active timeout value. The default is 20 seconds.

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               (1 << 0)
#define JOY_DNS_ON                 (1 << 1)
#define JOY_SSH_ON                 (1 << 2)
#define JOY_TLS_ON                 (1 << 3)
#define JOY_DHCP_ON                (1 << 4)
#define JOY_HTTP_ON                (1 << 5)
#define JOY_IKE_ON                 (1 << 6)
#define JOY_PAYLOAD_ON             (1 << 7)
#define JOY_EXE_ON                 (1 << 8)
#define JOY_ZERO_ON                (1 << 9)
#define JOY_RETRANS_ON             (1 << 10)
#define JOY_BYTE_DIST_ON           (1 << 11)
#define JOY_ENTROPY_ON             (1 << 12)
#define JOY_CLASSIFY_ON            (1 << 13)
#define JOY_HEADER_ON              (1 << 14)
#define JOY_PREMPTIVE_TMO_ON       (1 << 15)
#define JOY_IDP_ON                 (1 << 16)
#define JOY_IPFIX_EXPORT_ON        (1 << 17)
#define JOY_PPI_ON                 (1 << 18)
#define JOY_SALT_ON                (1 << 19)
```

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_initialize_no_config

```
int joy_initialize_no_config (void* config, FILE *err_info,
                              joy_init_t *data);
```

```
/*
 * Function: joy_initialize_no_config
 *
 * Description: This function initializes the Joy library
 *      to analyze the data features. This fucntion does not
 *      perform any configuration options as it is expetced that
 *      the caller did configuration prior to calling this function.
 *
 *      joy_initialize must be called before using any of the other
 *      API functions.
 *
 * Parameters:
 *      config - pointer to pre-setup config
 *      err_info - pointer to the file for error logging
 *      data - structure of Joy options
 *
 * Returns:
 *      0 - success
 *      1 - failure
 *
 */
```

This API is currently only used by joy.c as it sets up its configuration using command line parameters. It is recommended that new applications use the programmatic API joy_initialize() described above.

config

The **config** parameter contains the already setup configuration for the joy library and the application program.

err_info

The **err_info** parameter is the file descriptor used for sending data to the logfile or STDERR depending on the configuration.

data

The **data** parameter is the structure used to help configure the joy library. Since the configuration has mostly been done already and specified in the **config** parameter, this structure is only used for establishing the number of contexts to use and setting the active and inactive timeout values to something other than the defaults.

joy_print_config

```
int joy_print_config (uint8_t index, uint8_t format);
```

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

index

The **index** parameter determines which Joy context the printing of the configuration will occur in. Each context has its own output file so the index of the context is required.

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

```
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_update_compact_bd

```
int joy_update_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

```
int joy_label_subnets (char *label, uint8_t 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_update_ctx_global_time

```
void joy_update_ctx_global_time (uint8_t ctx_index,
                                 struct timeval *new_time);
```

```
/*
 * Function: joy_update_ctx_global_time
 *
 * Description: This function updates the global time of a given
 *      JOY library context. This is useful is adjusting the expiration
 *      of flow records when packets are not submitted for processing.
 *
 * Parameters:
 *      ctx_index - the index number of the JOY context
 *      new_time - pointer to the timeval structure containin the new time
 *
 * Returns:
 *      none.
 */
```

ctx_index

The **ctx_index** parameter is which library context to use when updating the global time. Each context has its own representation of global time usually based on when packets are processed.

new_time

The **new_time** parameter contains the new time to set the context’s global time value to.

joy_packet_to_context

```
uint8_t joy_packet_to_context (const unsigned char *packet
                               uint8_t num_contexts);
```

```
/*
 * Function: joy_packet_to_context
 *
 * Description: This function takes in an IP packet and using the
 *      5-tuple determines which context it should be sent to for
 *      data feature processing. This APi is useful for applications
 *      that use the JOY library and want to use the libraries default
 *      scheme for dividing up traffic among various worker contexts.
 *
 * Parameters:
 *      packet - pointer to the IP packet data
 *      num_contexts – number of contexts to use for the distribution
 *
 * Returns:
 *      context - the context number the packet belongs to for
 *                JOY processing.
 *          This algorithm keeps bidirectional flows in the same context.
 *
 */
```

packet

The **packet** parameter is the data packet the application wishes to process. The packet header will be parsed enough in order to determine the 5-tuple and then the context calculation is done.

num_context

The **num_context** parameter is the number of worker contexts to use when deciding where the packet should be sent based on the 5-tuple of the packet. The number of contexts is not sanity checked against any initialized value. This function is meant to be used a utility function prior to sending data into the JOY library. This function can server as a default distribution scheme for spreading work across multiple threads based on the packet 5-tuple.

joy_index_to_context

```
uint8_t joy_index_to_context (uint8_t ctx_index);
```

```
/*
 * Function: joy_index_to_context
 *
 * Description: This function takes in an index and returns a
 *      pointer to the context.
 *
 * Parameters:
 *      ctx_index - index of the context you want
 *
 * Returns:
 *      context - pointer to the context structure
 *
 */
```

ctx_index

The **ctx_index** parameter is the index number of the context the caller wishes to access.

joy_process_packet

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

```
/*
 * Function: joy_process_packet
 *
 * Description: This function invoked the packet processing function
 *      however, the application is permitted store some small amount
 *      of data in the flow record once it is created. This can be
 *      useful on the back end when an application wants to associate
 *      some data with a flow record during processing of the flow record.
 *
 * Parameters:
 *      ctx_index - index of the thread context to use
 *      header - libpcap header which contains timestamp, cap length
 *               and length
 *      packet - the actual data packet
 *      app_data_len - length of the application specific data
 *      app_data - pointer to the application data
 *
 * Notes:
 *      The application specific data length and data will be stored
 *      in the flow record. The application data is copied, so the calling
 *      application is responsible for freeing the data buffer, if necessary,
 *      when this function returns.
 *
 * Returns:
 *      Pointer to the flow record
 *
 */
```

ctx_index

The **ctx_index** parameter is signals which library context to use when processing this packet. This can simply be specific as:

```
    joy_ctx_id = 2;
    joy_process_packet(unsigned char *)joy_ctx_id, header, packet, 0, NULL);
```

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) */
    };
```

This parameter can be set to NULL, in which case the JOY Library will create a header structure on the fly with the current timestamp and length of the packet.

packet

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

app_data_len

The **app_data_len** parameter is the length of the application specific data that the caller wants to store in the flow record. The maximum number of bytes an application can store into a flow record is 32. If no application data is desired, then specify 0 here.

app_data

The **app_data** parameter is a pointer to the application specific data. App_data_len bytes will be copied from this pointer into the flow record so that the application can access this information at a later time. If no application data is desired, then this parameter should be set to NULL.

joy_libpcap_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_libpcap_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 thread 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_libpcap_process_packet, (unsigned char*)index);
```

This results in the joy_libpcap_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_libpcap_process_packet, then you would simply provide the context index as the first parameter in the API as such:

```
joy_ctx_id = 2;
joy_process_packet(unsigned char *)joy_ctx_id, 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) */
};
```

This gets filled in by libpcap automatically.

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 (uint8_t index, JOY_FLOW_TYPE 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 (uint8_t index, JOY_FLOW_TYPE 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.

joy_get_feature_counts

```
void joy_get_feature_counts (uint8_t index,
                             joy_ctx_feat_count_t *feat_counts);
```

```
/*
 * Function: joy_get_feature_counts
 *
 * Description: This function is pulls the record count for each
 *      data feature that is ready for a given context.
 *
 * Parameters:
 *      index - index of the context to use
 *      feat_counts - structure containing the record counts of each
 *                    feature ready
 *
 * Returns:
 *      none
 *
 */
```

index

The **index** parameter determines which library context to report the feature counts from.

feat_counts

The **feat_counts** parameter provides a place to store feature ready record counts for the given context. The feat_count parameter is a pointer to a structure with the following format:

```
     /* structure to hold feature ready counts for reporting */
     typedef struct joy_ctx_feat_count {
         uint16_t idp_recs_ready;
         uint16_t tls_recs_ready;
         uint16_t splt_recs_ready;
         uint16_t salt_recs_ready;
         uint16_t bd_recs_ready;
     } joy_ctx_feat_count_t;
```

joy_idp_external_processing

```
void joy_idp_external_processing (uint8_t index,
                                  joy_flow_rec_callback callback_fn);
```

```
/*
 * Function: joy_idp_external_processing
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the processing of the flow records
 *      that have IDP information ready for export.
 *      This function simply goes through the flow records and invokes
 *      the callback function to process the records that have IDP data.
 *
 *      Records that get processed have the IDP processed flag updated
 *      but are NOT removed from the flow record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      callback - function that actually does the flow record processing
 *
 * Returns:
 *      none
 *
 * Notes:
 *      The callback function will get passed a pointer to the flow record.
 *      The data_len and data fields will be ZERO and NULL respectively. This
 *      is because the IDP data can be retrieved directly from the flow
 *      record.
 */
```

index

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

callback_fn

The **callback_fn** parameter determines which parent application function is called to handle the actual processing of the flow record. The callback function will receive a pointer to the flow record upon invocation. Data_len will be 0 and data will be NULL for this API callback.

```
typedef void (joy_flow_rec_callback)(void *rec, unsigned int data_len,
unsigned char *data);
```

joy_tls_external_processing

```
void joy_tls_external_processing (uint8_t index,
                                  joy_flow_rec_callback callback_fn);
```

```
/*
 * Function: joy_tls_external_processing
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the processing of the flow records
 *      that have TLS information ready for export.
 *      This function simply goes through the flow records and invokes
 *      the callback function to process the records that have TLS data.
 *
 *      Records that get processed have the TLS processed flag updated
 *      but are NOT removed from the flow record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      callback - function that actually does the flow record processing
 *
 * Returns:
 *      none
 *
 * Notes:
 *      The callback function will get passed a pointer to the flow record.
 *      The data_len and data fields will be ZERO and NULL respectively. This
 *      is because the TLS data can be retrieved directly from the flow
 *      record.
 */
```

index

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

callback_fn

The **callback_fn** parameter determines which parent application function is called to handle the actual processing of the flow record. The callback function will receive a pointer to the flow record upon invocation. Data_len will be 0 and data will be NULL for this API callback.

```
    typedef void (joy_flow_rec_callback)(void *rec, unsigned int data_len,
    unsigned char *data);
```

joy_splt_external_processing

```
void joy_splt_external_processing (uint8_t index,
                                   JOY_EXPORT_TYPE export_frmt,
                                   unsigned int min_pkts,
                                   joy_flow_rec_callback callback_fn);
```

```
/*
 * Function: joy_splt_external_processing
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the processing of the flow records
 *      that have SPLT information ready for export.
 *      This function simply goes through the flow records and invokes
 *      the callback function to process the records that have SPLT data.
 *
 *      Records that get processed have the SPLT processed flag updated
 *      but are NOT removed from the flow record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      export_frmt - formatting of the exported data
 *      min_pkts - minimum number of packets processed before export occurs
 *      callback - function that actually does the flow record processing
 *
 * Returns:
 *      none
 *
 * Notes:
 *      The callback function will get passed a pointer to the flow record.
 *      The SPLT data needs to be preprocessed for export and as such, the
 *      data_len field will be the length of the preprocessed data and the
 *      data field will be a pointer to the actual preprocessed data.
 *      The callback does not need to worry about freeing the memory
 *      associated with the data.
 *      Once control returns from the callback function, the library will
 *      handle that memory. IF the callback function needs access to this
 *      data after it returns control to the library, then it should copy
 *      that data for later use.
 *
 *      For NetFlow V9:
 *           Data Length returned is always 40 bytes
 *                    (10 records times 4 bytes per record)
 *           If actual number of records is less than 10, padding occurs
 *      For IPFix:
 *           Data Length returned will be N * 4
 *                    (N records times 4 bytes per record)
 *              Maximum value for N is 10
 *           If actual number of records is less than 10, NO padding occurs
 *      Format of the Data (NetFlow V9 & IPFix):
 *           All length values (16-bits) followed by all times (16-bits)
 *           ie: for data length of 20 bytes
 *             format: len,len,len,len,len,time,time,time,time,time
 */
```

index

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

export_frmt

The **export_frmt** parameter determines whether Netflow v9 or IPFix formatting is used when filling in the SPLT data.

min_pkts

The **min_pkts** parameter allows the calling application to specify the minimum number of packets it will accept for considering the SPLT data valid.

callback_fn

The **callback_fn** parameter determines which parent application function is called to handle the actual processing of the flow record. The callback function will receive a pointer to the flow record upon invocation. Data_len will contain the number of bytes that data contains for the SPLT values. Data will contain the SPLT values.

```
    typedef void (joy_flow_rec_callback)(void *rec, unsigned int data_len,
    unsigned char *data);
```

joy_salt_external_processing

```
void joy_salt_external_processing (uint8_t index,
                                    JOY_EXPORT_TYPE export_frmt,
                                    unsigned int min_pkts,
                                    joy_flow_rec_callback callback_fn);
```

```
/*
 * Function: joy_salt_external_processing
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the processing of the flow records
 *      that have SALT information ready for export.
 *      This function simply goes through the flow records and invokes
 *      the callback function to process the records that have SPLT data.
 *
 *      Records that get processed have the SALT processed flag updated
 *      but are NOT removed from the flow record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      export_frmt - formatting of the exported data
 *      min_pkts - minimum number of packets processed before export occurs
 *      callback - function that actually does the flow record processing
 *
 * Returns:
 *      none
 *
 * Notes:
 *      The callback function will get passed a pointer to the flow record.
 *      The SALT data needs to be preprocessed for export and as such, the
 *      data_len field will be the length of the preprocessed data and the
 *      data field will be a pointer to the actual preprocessed data.
 *      The callback does not need to worry about freeing the memory
 *      associated with the data. Once control returns from the callback
 *      function, the library will free that memory. IF the callback function
 *      needs access to this data after it returns control to the library,
 *      then it should copy that data for later use.
 *
 *      For NetFlow V9:
 *           Data Length returned is always 40 bytes
 *                     (10 records times 4 bytes per record)
 *           If actual number of records is less than 10, padding occurs
 *      For IPFix:
 *           Data Length returned will be N * 4
 *                     (N records times 4 bytes per record)
 *              Maximum value for N is 10
 *           If actual number of records is less than 10, NO padding occurs
 *      Format of the Data (NetFlow V9 & IPFix):
 *           All length values (16-bits) followed by all times (16-bits)
 *           ie: for data length of 20 bytes
 *             format: len,len,len,len,len,time,time,time,time,time
 */
```

index

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

export_frmt

The **export_frmt** parameter determines whether Netflow v9 or IPFix formatting is used when filling in the SALT data.

min_pkts

The **min_pkts** parameter allows the calling application to specify the minimum number of packets it will accept for considering the SALT data valid.

callback_fn

The **callback_fn** parameter determines which parent application function is called to handle the actual processing of the flow record. The callback function will receive a pointer to the flow record upon invocation. Data_len will contain the number of bytes that data contains for the SALT values. Data will contain the SPAT values.

```
    typedef void (joy_flow_rec_callback)(void *rec, unsigned int data_len,
    unsigned char *data);
```

joy_bd_external_processing

```
void joy_bd_external_processing (uint8_t index,
                                 unsigned int min_octets,
                                 joy_flow_rec_callback callback_fn);
```

```
/*
 * Function: joy_bd_external_processing
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the processing of the flow records
 *      that have BD (byte distribution) information ready for export.
 *      This function simply goes through the flow records and invokes
 *      the callback function to process the records that have SPLT data.
 *
 *      Records that get processed have the BD processed flag updated
 *      but are NOT removed from the flow record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      min_octets - minimum number of octets processed before ready
 *      callback - function that actually does the flow record processing
 *
 * Returns:
 *      none
 *
 * Notes:
 *      The callback function will get passed a pointer to the flow record.
 *      The BD data needs to be preprocessed for export and as such, the
 *      data_len field will be the length of the preprocessed data and the
 *      data field will be a pointer to the actual preprocessed data. The
 *      callback does not need to worry about freeing the memory associated
 *      with the data. Once control returns from the callback function, the
 *      library will free that memory. IF the callback function needs access
 *      to this data after it returns control to the library, then it should
 *      copy that data for later use.
 *
 *      For NetFlow V9 and IPFix:
 *          The data length is always 512 bytes. Currently only BD format
 *              uncompressed is defined in the spec.
 *          The data format is a series of 16-bit values representing the
 *              count of a given ascii value. The first 16-bit value
 *              represents the number of times ascii value 0 was seen in the
 *              flow. The second 16-bit value represents the number times the
 *              ascii value 1 was seen in the flow.
 *              This continues for all ascii values up to value 255.
 */
```

index

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

min_octets

The **min_octets** parameter allows the calling application to specify the minimum number of octets it will accept for considering the BD data valid.

callback_fn

The **callback_fn** parameter determines which parent application function is called to handle the actual processing of the flow record. The callback function will receive a pointer to the flow record upon invocation. Data_len will contain the number of bytes that data contains for the BD values. Data will contain the BD values.

```
typedef void (joy_flow_rec_callback)(void *rec, unsigned int data_len,
unsigned char *data);
```

Cleanup APIs

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

joy_delete_flow_records

```
unsigned int joy_delete_flow_records (uint8_t index,
                                      unsigned int cond_bitmask);
```

```
/*
 * Function: joy_delete_flow_records
 *
 * Description: This function is allows the calling application of
 *      the Joy library to handle the explicit deletion of flow records
 *      from the flow_record list.
 *
 * Parameters:
 *      index - index of the context to use
 *      cond_bitmask - bitmask of conditions on which records to delete
 *             (JOY_IDP_PROCESSED, JOY_TLS_PROCESSED, JOY_SALT_PROCESSED,
 *              JOY_SPLT_PROCESSED, JOY_BD_PROCESSED, JOY_ANY_PROCESSED)
 *
 * Returns:
 *      unsigned int - number of records deleted
 *
 */
```

index

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

cond_bitmask

The cond_bitmask parameter specifies under which conditions the flow record should be removed. The following values can be applied to the cond_bitmask:

```
#define JOY_DELETE_ALL          (0)
#define JOY_IDP_PROCESSED       (1 << 0)
#define JOY_TLS_PROCESSED       (1 << 1)
#define JOY_SALT_PROCESSED      (1 << 2)
#define JOY_SPLT_PROCESSED      (1 << 3)
#define JOY_BD_PROCESSED        (1 << 4)
```

These values can be combined to assess multiple conditions. For instance, if you wanted to delete records that had both IDP and TLS processed you could set the **cond_bitmask** to (JOY_IDP_PROCESSED | JOY_TLS_PROCESSED).

joy_purge_old_flow_records

```
unsigned int joy_purge_old_flow_records (uint8_t index,
                                         unsigned int rec_age);
```

```
/*
 * Function: joy_purge_old_flow_records
 *
 * Description: This function allows the calling application of
 *      the Joy library to handle the forced removal of flow records
 *      that are older than the time value passed in by the caller.
 *
 * Parameters:
 *      index - index of the context to use
 *      rec_age - age of the records in seconds
 *
 * Returns:
 *      unsigned int - number of records deleted
 *
 */
```

index

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

rec_age

The rec_age parameter specifies in seconds how old the records to be purged need to be.

joy_context_cleanup

```
void joy_context_cleanup (uint8_t index);
```

```
/*
 * Function: joy_context_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 with this context anymore. This API will flush out any remaining data and then free up the memory structures for the flow records.

joy_shutdown

```
void joy_shutdown (void);
```

```
/*
 * Function: joy_shutdown
 *
 * Description: This function cleans up the JOY library and essentially
 *      shuts the library down and reverts back to clean unused state.
 *
 * Parameters:
 *      none
 *
 * Returns:
 *      none
 *
 */
```

This API should only be called when you are finished using the Joy library completely. This API will free up any memory associated with the JOY Library and also put the library back into an uninitialized state.