PySiLK: SiLK in Python

Release 1.1.0

CERT NetSA

July 8, 2008

silk-help@cert.org

Abstract

This document describes how to read and write SiLK packed data from within Python.

Contents

1 `silk` — SiLK record and file support

The `silk` module supplies objects for interfacing with SiLK records and data files.

The `silk` module exports the following functions:

`init_site`(*filename*)

Uses the given *filename* as the site file. If *filename* is omitted, the value of $SILK_CONFIG_FILE will be used as the name of the configuration file. If $SILK_CONFIG_FILE is not set, the module looks for a file named 'silk.conf' in the following directories: the directory specified in the $SILK_DATA_ROOTDIR environment variable; the data root directory that is compliled into SiLK; the directories '$SILK_PATH/share/silk/', and '$SILK_PATH/share/'.

This function should not generally be called explicitly unless one wishes to use a non-default site configuration file.

The `init_site` function can only be called once. Subsequent invocations will throw a `RuntimeError`.

Several functions and use of class members in PySilk implicitly call `init_site` if it has not already been called. These include `sensors`, `classtypes`, `classes`, `have_site_config`, the `RWRec.as_dict` method, and using the `RWRec` attributes `classname`, `typename`, `classtype`, or `sensor`.

sensors()
Returns a tuple of valid sensor names. Implicitly calls `init_site` with no argument if it has not yet been called.

classtypes()
Returns a tuple of valid (class name, type name) tuples. Implicitly calls `init_site` with no argument if it has not yet been called.

classes()
Returns a tuple of valid class names. Implicitly calls `init_site` with no argument if it has not yet been called.

ipv6_enabled()
Return `True` if SiLK was compiled with IPv6 support, `False` otherwise.

initial_tcpflags_enabled()
Return `True` if SiLK was compiled with support for initial TCP flags, `False` otherwise.

have_site_config()
Returns `True` if the module was able to locate the SiLK configuration file, `False` otherwise. Implicitly calls `init_site` with no argument if it has not yet been called.

1.1 Available Types

class IPAddr
A representation for IP Addresses.

class IPWildcard
A representation of IP wildcard addresses or CIDR blocks.

class IPSet
A representation of an IPset.

class TCPFlags
A representation of TCP flags.

class RWRec
A representation of a SiLK data record.

class FGlob
An iterable object that allows retrievel of filenames in a SiLK data store.

1.2 PySilk Example

The following is an example using the PySiLK bindings. The code is meant to show some standard PySiLK techniques, but is not otherwise meant to be useful. Explanations for the code can be found inline in the comments.

```
#!/usr/bin/python2.4

# Import the pysilk bindings
from silk import *

# Import sys for the command line arguments.
import sys

# Main function
def main():

    if len(sys.argv) != 3:
        print ("Usage: %s infile outset" % sys.argv[0])

    # Open an silk file for reading
    infile = SilkFile(sys.argv[1], READ)

    # Create an empty IPset
    destset = IPSet()

    # Loop over the records in the file
    for rec in infile:

      # Do comparisons based on rwrec field value
      if (rec.protocol == 6 and rec.sport in [80, 8080] and
          rec.packets > 3 and rec.bytes > 120):

          # Add the dest IP of the record to the IPset
          destset.add(rec.dip)


    # Save the IPset for future use
    destset.save(sys.argv[2])


    # count the items in the set
    count = 0
    for addr in destset:
        count = count + 1

    print "%d addresses" % count

    # Another way to do the same
    print "%d addresses" % len(destset)


    # Print the ip blocks in the set
    for base_prefix in destset.cidr_iter():
        print "%s/%d" % base_prefix

# Call the main() function when this program is started
if __name__ == '__main__':
    main()
```

1.3 `IPAddr` Objects

An `IPAddr` object represents an IPv4 or IPv6 address.

class `IPAddr`(*address*)

The constructor takes either a string *address*, which must be a string representation of either an IPv4 or IPv6 address, or an integer representation of the address. IPv6 addresses are only accepted if `ipv6_enabled()` returns `True`.

Examples:

```
>>> addr1 = IPAddr('192.160.1.1')
>>> addr2 = IPAddr('2001:db8::1428:57ab')
>>> addr3 = IPAddr('::ffff:12.34.56.78')
>>> addr4 = IPAddr(0xffffffff)
>>> addr5 = IPAddr(0xffffffffffffffffffffffffffffffff)
```

Supported operations:

Operation	Result
addr1 `<` *addr2*	*addr1* is considered less than *addr2* if the 128-bit representation of *addr1* is less than *addr2*
`int(`*addr1*`)`	The integer representation of *addr*

Instance methods:

`ipv6`()

Return `True` if the address is an IPv6 address, `False` otherwise.

`__str__`()

For an address *addr*, `str(`*addr*`)` returns a human-readable representation of that address.

1.4 `IPWildcard` Objects

An `IPWildcard` object represents a range or block of IP addresses. The `IPWildcard` object handles iteration over IP addresses with `for` *x* `in` *wildcard*.

class `IPWildcard`(*wildcard*)

The constructor takes a string representation *wildcard* of the wildcard address.

The string *wildcard* can be in CIDR notation, an integer, an integer with a CIDR designation, or an entry in SiLK wildcard notation. In SiLK wildcard notation, a wildcard is represented as a string IP address in canonical form with an `x` representing an entire octet or hexadectet. An IP wildcard string can also have lists or ranges in place of an octet or hexadectet. IPv6 wildcard addresses are only accepted if `ipv6_enabled()` returns `True`.

Examples:

```
>>> a = IPWildcard('1.2.3.0/24')
>>> b = IPWildcard('ff80::/16')
>>> c = IPWildcard('1.2.3.4')
>>> d = IPWildcard('::FFFF:0102:0304')
>>> e = IPWildcard('16909056')
>>> f = IPWildcard('16909056/24')
>>> g = IPWildcard('1.2.3.x')
>>> h = IPWildcard('1:2:3:4:5:6:7.x')
>>> i = IPWildcard('1.2,3.4,5.6,7')
>>> j = IPWildcard('1.2.3.0-255')
>>> k = IPWildcard('::2-4')
>>> l = IPWildcard('1-2:3-4:5-6:7-8:9-a:b-c:d-e:0-ffff')
```

Supported operations:

Operation	Result
addr `in` *wildcard*	`True` if *addr* is in *wildcard*, `False` otherwise
addr `not in` *wildcard*	`False` if *addr* is in *wildcard*, `True` otherwise
string `in` *wildcard*	Same as: `IPAddr(`*string*`) in` *wildcard*
string `not in` *wildcard*	Same as: `IPAddr(`*string*`) not in` *wildcard*

Instance methods:

`__str__``()`
For an IP wildcard *wild*, `str(`*wild*`)` returns the string that was used to make the wildcard.

1.5 `IPSet` Objects

An `IPSet` object represents any set of IP addresses, as produced by **rwsetbuild** and related programs. The `IPSet` object handles iteration over IP addresses with `for` *x* `in` *set*, and iteration over CIDR blocks using `for` *x* `in` *set*`.cidr_iter().`

class `IPSet``(`*[iterable]*`)`
The constructor creates an empty IPset. If an iterable is supplied as an argument, each item of the iterable will be added to the IPset. Each item of the iterable should either be an IPv4 `IPAddr` or a string representing a valid IPv4 address.

Other constructors, all class methods:

`load``(`*path*`)`
Creates an `IPSet` from an IPset saved in a file. *path* must be a valid location of an IPset.

Supported operations:

Operation	Equivalent	Result	Notes
`len(`*s*`)`		cardinality of IPset *s*	(1)
s`.cardinality()`		cardinality of IPset *s*	
addr `in` *s*		test *addr* for membership in *s*	(2)
addr `not in` *s*		test *addr* for non-membership in *s*	(2)
s`.issubset(`*t*`)`	*s* `<=` *t*	test whether every element in *s* is in *t*	(3)
s`.issuperset(`*t*`)`	*s* `>=` *t*	test whether every element in *t* is in *s*	(3)
s`.union(`*t*`)`	*s* `|` *t*	new IPset with elements from both *s* and *t*	(3)
s`.intersection(`*t*`)`	*s* `&` *t*	new IPset with elements common to *s* and *t*	(3)
s`.difference(`*t*`)`	*s* `-` *t*	new IPset with elements in *s* but not in *t*	(3)
s`.symmetric_difference(`*t*`)`	*s* `^` *t*	new IPset with elements in either *s* or *t* but not both	(3)
s`.copy()`		new set with a copy of *s*	
s`.update(`*t*`)`	*s* `|=` *t*	update *s*, adding elements from *t*	(3)
s`.intersection_update(`*t*`)`	*s* `&=` *t*	update *s*, keeping only elements found in both *s* and *t*	(3)
s`.difference_update(`*t*`)`	*s* `-=` *t*	update *s*, removing elements found in *t*	(3)
s`.symmetric_difference_update(`*t*`)`	*s* `^=` *t*	update *s*, keeping elements found in *s* or *t* but not in both	(3)
s`.add(`*addr*`)`		add element *addr* to IPset *s*	(2)
s`.remove(`*addr*`)`		remove *addr* from IPset *s*; raises `KeyError` if not present	
s`.discard(`*addr*`)`		removes *addr* from IPset *s* if present	
s`.clear()`		remove all elements from IPset *s*	

Notes:

(1) May throw `OverflowError` if there are too many IP addresses in the IPset. Use *s*`.cardinality()` instead.

(2) *addr* can be an `IPAddr`, an `IPWildcard`, or the string representation of either. The address or addresses must be an IPv4 addresses.

(3) With the non-operator version of this method, *t* can be any iterable object of IP addresses or IP address strings. The operator version requires that *t* be an `IPSet`.

Instance methods:

`cidr_iter``()`
Returns an iterator over CIDR blocks. Each iteration returns a tuple, the first element of which is the first IP address in the block, the second of which is the prefix length of the block. Can be used as `for (`*addr*`, `*prefix*`) in `*s*`.cidr_iter().`

`save`(*filename*)
Saves the IPSet in the file *filename*.

1.6 `TCPFlags` Objects

A `TCPFlags` object represents the eight bits of flags from a TCP session.

class `TCPFlags`(*value*)
The constructor takes either a TCPFlags value, a string, or an integer. If a TCPFlags value, it returns a copy of that value. If an integer, the integer should represent the 8-bit representation of the flags. If a string, the string should consist of a concatenation of zero or more of the characters '`F`', '`S`', '`R`', '`P`', '`A`', '`U`', '`E`', and '`C`'—upper or lower-case—representing the FIN, SYN, RST, PSH, ACK, URG, ECE, and CWR flags. Spaces in the string are ignored.

Examples:

```
>>> a = TCPFlags('SA')
>>> b = TCPFlags(5)
```

Instance attributes (read-only):

Attribute	Value
`FIN`	`True` if the FIN flag is set, `False` otherwise
`SYN`	`True` if the SYN flag is set, `False` otherwise
`RST`	`True` if the RST flag is set, `False` otherwise
`PSH`	`True` if the PSH flag is set, `False` otherwise
`ACK`	`True` if the ACK flag is set, `False` otherwise
`URG`	`True` if the URG flag is set, `False` otherwise
`ECE`	`True` if the ECE flag is set, `False` otherwise
`CWR`	`True` if the CWR flag is set, `False` otherwise

Supported operations:

Operation	Result
f	The bitwise inversion (not) of *f*
f1 `&` *f2*	The bitwise intersection (and) of the flags from *f1* and *f2*
f1 `|` *f2*	The bitwise union (or) of the flags from *f1* and *f2*
f1 `^` *f2*	The bitwise exclusive disjunction (xor) of the flags from *f1* and *f2*
`int`(*f*)	The integer value of the flags *f*
f	Can be used as a truth value with any flag set == `True`, `False` otherwise

Constants:

The following constants are defined:

Constant	Meaning
FIN	A TCPFlags value with only the FIN flags set
SYN	A TCPFlags value with only the SYN flags set
RST	A TCPFlags value with only the RST flags set
PSH	A TCPFlags value with only the PSH flags set
ACK	A TCPFlags value with only the ACK flags set
URG	A TCPFlags value with only the URG flags set
ECE	A TCPFlags value with only the ECE flags set
CWR	A TCPFlags value with only the CWR flags set

Supported methods:

matches (*flagmask*)
Given a *flagmask* of the form "*flags/mask*", returns True if if the flags of self match *flags* after being masked with *mask*, False otherwise.

Given a *flagmask* without the '/', checks for literal equality, as if the mask contained all flags.

__str__()
For an TCPFlags object *f*, str(*f*) returns the a string representation of the flags set in *f*.

1.7 RWRec Objects

An RWRec object represents a SiLK record.

class RWRec (*[rec],[field=value],...*)
This constructor creates an empty RWRec object. If an RWRec *rec* is supplied, it will create a copy of *rec*. The variable *rec* can be a dictionary, such as that supplied by RWRec.as_dict(). Initial values for record fields can be included.

Example:

```
>>> recA = RWRec(input=10, output=20)
>>> recB = RWRec(recA, output=30)
>>> (recA.input, recA.output)
(10, 20)
>>> (recB.input, recB.output)
(10, 30)
```

Instance attributes:

Attribute	Value	Type
`application`	The "service" port set by the collector	integer
`bytes`	The count of the number of bytes in the flow	integer
`classname`	The class name of the record (read-only) (1)	string
`classtype`	A tuple of the class name and type name of the record (1)	(string, string)
`dip`	The destination IP (can be set as a string)	`IPAddr`
`dport`	The destination port	integer
`duration`	The duration of the flow	`datetime.timedelta`
`etime`	The end time of the flow	`datetime.datetime`
`initflags`	The TCP flags of the first packet of the flow (may be `None`)	`TCPFlags`
`icmpcode`	The ICMP code (only valid if `protocol` is `1`)	integer
`icmptype`	The ICMP type value (only valid if `protocol` is `1`)	integer
`input`	The router's incoming SNMP interface	integer
`nhip`	The router's next-hop IP (can be set as a string)	`IPAddr`
`output`	The router's outgoing SNMP interface	integer
`packets`	The packet cout for the flow	integer
`protocol`	The IP protocol	integer
`restflags`	The union of the flags of all but the first packet of the flow (may be `None`)	`TCPFlags`
`sensor`	The sensor ID (1)	string
`sip`	The source IP (can be set as a string)	`IPAddr`
`sport`	The source port	integer
`stime`	The start time of the flow	`datetime.datetime`
`tcpflags`	The union of the TCP flags of all packets in the flow	`TCPFlags`
`timeout_killed`	Whether the flow ended early due to timeout by the collector (may be `None`)	boolean
`timeout_started`	Whether the flow is a contination from a timed-out flow (may be `None`)	boolean
`typename`	The type name of the record (read-only) (1)	string

Notes:

(1) Using or setting this attribute implicitly calls `init_site` with no argument if it has not yet been called.

Supported methods:

`is_web``()`
: `True` if the record can be represented as a web record, `False` otherwise.

`as_dict``()`
: Returns a dictionary representing the contents of the record. Implicitly calls `init_site` with no argument if it has not yet been called.

`__str__``()`
: For an record *rec*, `str(`*rec*`)` returns the string representation of *rec*`.as_dict()`.

Supported operations:

Operation	Result
rec1 `==` *rec2*	True if *rec1* is structurally equivalent to *rec2*
rec1 `!=` *rec2*	True if *rec1* is not structurally equivalent to *rec2*

1.8 `SilkFile` Object

An `SilkFile` object represents a channel for writing to or reading from SiLK flow files. A SiLK file open for reading can be iterated over using `for` *rec* `in` *file*.

class `SilkFile` (*filename, mode, compression=DEFAULT, notes=[], invocations=[]*)

The constructor takes a filename, a mode, and a set of optional keyword parameters. The *filename* should be the path to the file to open. The *mode* should be one of the following constant values:

Mode	Meaning
`READ`	Open file for reading
`WRITE`	Open file for writing
`APPEND`	Open file for appending

A few filenames are treated specially. The filename 'stdin' maps to the standard input stream when the mode is `READ`. The filenames 'stdout' and 'stderr' map to the standard output and standard error streams respectively when the mode is `WRITE`. A filename consisting of a single hyphen ('-') maps to the standard input if the mode is `READ`, and to the standard output if the mode is `WRITE`.

The *compression* parameter can be one of the following constants:

Constant	Meaning
`DEFAULT`	Default compression scheme compiled into SiLK
`NO_COMPRESSION`	No compression
`ZLIB`	Use zlib block compression
`LZO1X`	Use lzo1x block compression

If *notes* or *invocations* are set, they should be list of strings. These add annotation and invocation headers to the file.

Examples:

```
>>> myinputfile = SilkFile('/path/to/file', READ)
>>> myoutputfile = SilkFile('/path/to/file', WRITE, compression=LZO1X,
                            notes=['My output file',
                                   'another annotation'])
```

Instance methods:

`read`()

Returns an `RWRec` representing the next record in the `SilkFile`. If there are no records left in the file, returns `None`.

`write`(*rec*)

Writes the `RWRec` *rec* to the file. Returns `None`.

`next`()

A `SilkFile` object is its own iterator, for example `iter`(*f*) returns *f*. When the `SilkFile` is used as an iterator, the `next()` method is called repeatedly. This method returns the next record, or raises `StopIteration` when EOF is hit.

`notes`()

Returns the list of annotation headers for the file as a list of strings.

`invocations`()

Returns the list of invocation headers for the file as a list of strings.

`close`()

Closes the file. Returns `None`.

1.9 `FGlob` Objects

An `FGlob` object is an iterable object which iterates over filenames from a SiLK data store. It does this internally by calling the **rwfglob** program. The FGlob object assumes that the **rwfglob** program is in the PATH, and will throw an execption when used if not.

class FGlob(*classname=None, type=None, sensors=None, start_date=None, end_date=None, data_rootdir=None, site_config_file=None*)

Arguments are:

classname, if given, should be a string representing the class name. If not given, defaults based on the site configuration file.

type, if given, can be either a string representing a type name or comma-separated list of type names, or can be a list of strings representing type names. If not given, defaults based on the site configuration file.

sensors, if given, should be either a string representing a comma-separated list of sensor names or IDs, and integer representing a sensor ID, or a list of strings or integers representing sensor names or IDs. If not given, defaults to all sensors.

start_date, if given, should be either a string in the format `YYYY/MM/DD[:HH]`, a date object, a datetime object (which will be used to the precision of one hour), or a time object (which is used for the given hour on the current date). If not given, defaults to start of current day.

end_date, if given, should be either a string in the format `YYYY/MM/DD[:HH]`, a date object, a datetime object (which will be used to the precision of one hour), or a time object (which is used for the given hour on the current date). If not given, defaults to *start_date*. *end_date* cannot be used without a *start_date*.

data_rootdir, if given, should be a string representing the directory in which to find the packed SiLK data files. If not given, defaults to $SILK_DATA_ROOTDIR or the compiled-in default.

site_config_file, if given, should be a string representing the path of the site configuration file. If not given, defaults to $SILK_CONFIG_FILE or '$SILK_DATA_ROOTDIR/silk.conf'.

At least one of *classname*, *type*, *sensors*, *start_date* must be specified.

An `FGlob` object can be used as a standard iterator. For example:

```
for filename in FGlob(classname="all", start_date="2005/09/22"):
    for rec in SilkFile(filename):
        ...
```