Introduction

The PySiLK Reference Guide contains the `pysilk(3)` and `silkpython(3)` manual pages in a single document. The manual page for every SiLK tool is available in the SiLK reference guide. The SiLK Analysis Handbook provides both a tutorial for learning about the tools and examples of how they can be used in analyzing flow data. See the SiLK Installation Handbook for instructions on installing SiLK at your site.
DESCRIPTION

This document describes the features of **PySiLK**, the SiLK Python extension. It documents the objects and methods that allow one to read, manipulate, and write SiLK Flow records, IPsets, Bags, and Prefix Maps (pmaps) from within **python(1)**. PySiLK may be used in a stand-alone Python script or as a plug-in from within the SiLK tools **rffielder(1)**, **rwcut(1)**, **rwgroup(1)**, **rwsort(1)**, **rwstats(1)**, and **rwuniq(1)**. This document describes the objects and methods that PySiLK provides; the details of using those from within a plug-in are documented in the **silkpython(3)** manual page.

The SiLK Python extension defines the following objects and modules:

- **IPAddr object**
  - Represents an IP Address.

- **IPv4Addr object**
  - Represents an IPv4 Address.

- **IPv6Addr object**
  - Represents an IPv6 Address.

- **IPWildcard object**
  - Represents CIDR blocks or SiLK IP wildcard addresses.

- **IPSet object**
  - Represents a SiLK IPset.

- **PrefixMap object**
  - Represents a SiLK Prefix Map.

- **Bag object**
  - Represents a SiLK Bag.

- **TCPFlags object**
  - Represents TCP flags.
**RWRec object**
Represents a SiLK Flow record.

**SilkFile object**
Represents a channel for writing to or reading from SiLK Flow files.

**FGlob object**
Allows retrieval of filenames in a SiLK data store. See also the **silk.site** module.

**silk.site module**
Defines several functions that relate to the SiLK site configuration and allow iteration over the files in a SiLK data store.

**silk.plugin module**
Defines functions that may only be used in SiLK Python plug-ins.

The SiLK Python extension provides the following functions:

**silk.get_configuration(name=None)**
When `name` is **None**, return a dictionary whose keys specify aspects of how SiLK was compiled. When `name` is provided, return the dictionary value for that key, or **None** when `name` is an unknown key. The dictionary’s keys and their meanings are:

- **COMPRESSION_METHODS**
  A list of strings specifying the compression methods that were compiled into this build of SiLK. The list will contain one or more of **NO_COMPRESSION**, **ZLIB**, **LZO1X**, and/or **SNAPPY**.

- **INITIAL_TCPFLAGS_ENABLED**
  True if SiLK was compiled with support for initial TCP flags; **False** otherwise.

- **IPV6_ENABLED**
  True if SiLK was compiled with IPv6 support; **False** otherwise.

- **SILK_VERSION**
  The version of SiLK linked with PySiLK, as a string.

- **TIMEZONE_SUPPORT**
  The string **UTC** if SiLK was compiled to use UTC, or the string **local** if SiLK was compiled to use the local timezone.

**Since SiLK 3.8.1.**

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

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

**silk.init_country_codes(filename=None)**
Initialize PySiLK’s country code database. `filename` should be the path to a country code prefix map, as created by `rwgeoip2ccmap(1)`. If `filename` is not supplied, SiLK will look first for the file specified by `$SILK_COUNTRY_CODES`, and then for a file named `country_codes.pmap` in `$SILK_PATH/share/silk`, `$SILK_PATH/share`, `/usr/local/share/silk`, and `/usr/local/share`. (The latter two assume that SiLK was installed in `/usr/local.`) Will throw a **RuntimeError** if loading the country code prefix map fails.

**silk.silk_version()**
Return the version of SiLK linked with PySiLK, as a string.
IPAddr Object

An IPAddr object represents an IPv4 or IPv6 address. These two types of addresses are represented by two subclasses of IPAddr: IPv4Addr and IPv6Addr.

class silk.IPAddr(address)

The constructor takes a string address, which must be a string representation of either an IPv4 or IPv6 address, or an IPAddr object. IPv6 addresses are only accepted if silk.ipv6_enabled() returns True. The IPAddr object that the constructor returns will be either an IPv4Addr object or an IPv6Addr object.

For compatibility with releases prior to SiLK 2.2.0, the IPAddr constructor will also accept an integer address, in which case it converts that integer to an IPv4Addr object. This behavior is deprecated. Use the IPv4Addr and IPv6Addr constructors instead.

Examples:

```python
>>> addr1 = IPAddr('192.160.1.1')
>>> addr2 = IPAddr('2001:db8::1428:57ab')
>>> addr3 = IPAddr('::ffff:12.34.56.78')
>>> addr4 = IPAddr(addr1)
>>> addr5 = IPAddr(addr2)
>>> addr6 = IPAddr(0x10000000) # Deprecated as of SiLK 2.2.0
```

Supported operations and methods:

Inequality Operations

In all the below inequality operations, whenever an IPv4 address is compared to an IPv6 address, the IPv4 address is converted to an IPv6 address before comparison. This means that IPAddr("0.0.0.0") == IPAddr("::ffff:0.0.0.0").

```python
addr1 == addr2
Return True if addr1 is equal to addr2; False otherwise.
```

```python
addr1 != addr2
Return False if addr1 is equal to addr2; True otherwise.
```

```python
addr1 < addr2
Return True if addr1 is less than addr2; False otherwise.
```

```python
addr1 <= addr2
Return True if addr1 is less than or equal to addr2; False otherwise.
```

```python
addr1 > addr2
Return True if addr1 is greater than addr2; False otherwise.
```

```python
addr1 >= addr2
Return True if addr1 is greater than or equal to addr2; False otherwise.
```

```python
addr.is_ipv6()
Return True if addr is an IPv6 address, False otherwise.
```

```python
addr.isipv6()
(DEPRECATED in SiLK 2.2.0) An alias for is_ipv6().
```
addr.to_ipv6()

If \textit{addr} is an \texttt{IPv6Addr}, return a copy of \textit{addr}. Otherwise, return a new \texttt{IPv6Addr} mapping \textit{addr} into the ::ffff:0/96 prefix.

addr.to_ipv4()

If \textit{addr} is an \texttt{IPv4Addr}, return a copy of \textit{addr}. If \textit{addr} is in the ::ffff:0/96 prefix, return a new \texttt{IPv4Addr} containing the IPv4 address. Otherwise, return \texttt{None}.

\texttt{int(addr)}

Return the integer representation of \textit{addr}. For an IPv4 address, this is a 32-bit number. For an IPv6 address, this is a 128-bit number.

\texttt{str(addr)}

Return a human-readable representation of \textit{addr} in its canonical form.

addr.padded()

Return a human-readable representation of \textit{addr} which is fully padded with zeroes. With IPv4, it will return a string of the form "xxx.xxx.xxx.xxx". With IPv6, it will return a string of the form "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx".

addr.octets()

Return a tuple of integers representing the octets of \textit{addr}. The tuple's length is 4 for an IPv4 address and 16 for an IPv6 address.

addr.mask(mask)

Return a copy of \textit{addr} masked by the IPAddr \textit{mask}.

When both addresses are either IPv4 or IPv6, applying the mask is straightforward.

If \textit{addr} is IPv6 but \textit{mask} is IPv4, \textit{mask} is converted to IPv6 and then the mask is applied. This may result in an odd result.

If \textit{addr} is IPv4 and \textit{mask} is IPv6, \textit{addr} will remain an IPv4 address if masking \textit{mask} with ::ffff:0000:0000 results in ::ffff:0000:0000, (namely, if bytes 10 and 11 of \textit{mask} are 0xFFFF). Otherwise, \textit{addr} is converted to an IPv6 address and the mask is performed in IPv6 space, which may result in an odd result.

addr.mask_prefix(prefix)

Return a copy of \textit{addr} masked by the high \textit{prefix} bits. All bits below the \textit{prefix}th bit will be set to zero. The maximum value for \textit{prefix} is 32 for an \texttt{IPv4Addr}, and 128 for an \texttt{IPv6Addr}.

addr.country_code()

Return the two character country code associated with \textit{addr}. If no country code is associated with \textit{addr}, return \texttt{None}. The country code association is initialized by the \texttt{silk.init_country_codes()} function. If \texttt{init_country_codes()} is not called before calling this method, it will act as if \texttt{init_country_codes()} was called with no argument.

\textbf{IPv4Addr Object}

An \texttt{IPv4Addr} object represents an IPv4 address. \texttt{IPv4Addr} is a subclass of \texttt{IPAddr}, and supports all operations and methods that \texttt{IPAddr} supports.
class silk.IPv4Addr(address)

The constructor takes a string address, which must be a string representation of IPv4 address, an IPAddr object, or an integer. A string will be parsed as an IPv4 address. An IPv4Addr object will be copied. An IPv6Addr object will be converted to an IPv4 address, or throw a ValueError if the conversion is not possible. A 32-bit integer will be converted to an IPv4 address.

Examples:

```python
>>> addr1 = IPv4Addr('192.160.1.1')
>>> addr2 = IPv4Addr(IPAddr('::ffff:12.34.56.78'))
>>> addr3 = IPv4Addr(addr1)
>>> addr4 = IPv4Addr(0x100000000)
```

IPv6Addr Object

An IPv6Addr object represents an IPv6 address. IPv6Addr is a subclass of IPAddr, and supports all operations and methods that IPAddr supports.

class silk.IPv6Addr(address)

The constructor takes a string address, which must be a string representation of either an IPv6 address, an IPAddr object, or an integer. A string will be parsed as an IPv6 address. An IPv6Addr object will be copied. An IPv4Addr object will be converted to an IPv6 address. A 128-bit integer will be converted to an IPv6 address.

Examples:

```python
>>> addr1 = IPAddr('2001:db8::1428:57ab')
>>> addr2 = IPv6Addr(IPAddr('192.160.1.1'))
>>> addr3 = IPv6Addr(addr1)
>>> addr4 = IPv6Addr(0x100000000000000000000000)
```

IPWildcard Object

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 silk.IPWidcard(wildcard)

The constructor takes a string representation wildcard of the wildcard address. The string wildcard can be an IP address, an IP with a 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 an IP address in canonical form with each octet (IPv4) or hexadectet (IPv6) represented by one of following: a value, a range of values, a comma separated list of values and ranges, or the character ‘x’ used to represent the entire octet or hexadectet. IPv6 wildcard addresses are only accepted if silk.ipv6_enabled() returns True. The wildcard element can also be an IPWildcard, in which case a duplicate reference is returned.

Examples:

```python
>>> a = IPWildcard('1.2.3.0/24')
>>> b = IPWildcard('ff80::/16')
```
```python
>>> 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:c:d-e:0-ffff')
>>> m = IPWildcard(a)
```

Supported operations and methods:

- `addr in wildcard`
  Return `True` if `addr` is in `wildcard`, `False` otherwise.

- `addr not in wildcard`
  Return `False` if `addr` is in `wildcard`, `True` otherwise.

- `string in wildcard`
  Return the result of `IPAddr(string) in wildcard`.

- `string not in wildcard`
  Return the result of `IPAddr(string) not in wildcard`.

- `wildcard.is_ipv6()`
  Return `True` if `wildcard` contains IPv6 addresses, `False` otherwise.

- `str(wildcard)`
  Return the string that was used to construct `wildcard`.

### IPSet Object

An IPSet object represents a set of IP addresses, as produced by `rwset(1)` and `rwsetbuild(1)`. 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()`.

In the following documentation, and `ip_iterable` can be any of:

- an `IPAddr` object representing an IP address
- the string representation of a valid IP address
- an `IPWildcard` object
- the string representation of an `IPWildcard`
- an iterable of any combination of the above
- another IPSet object
class silk.IPSet([ip_iterable])

The constructor creates an empty IPset. If an ip_iterable is supplied as an argument, each member of ip_iterable will be added to the IPset.

Other constructors, all class methods:

silk.IPSet.load(path)

Create an IPSet by reading a SiLK IPset file. path must be a valid location of an IPset.

Other class methods:

silk.IPSet.supports_ipv6()

Return whether this implementation of IPsets supports IPv6 addresses.

Supported operations and methods:

In the lists of operations and methods below,

- set is an IPSet object
- addr can be an IPAddr object or the string representation of an IP address.
- set2 is an IPSet object. The operator versions of the methods require an IPSet object.
- ip_iterable is an iterable over IP addresses as accepted by the IPSet constructor. Consider ip_iterable as creating a temporary IPSet to perform the requested method.

The following operations and methods do not modify the IPSet:

set.cardinality()

Return the cardinality of set.

len(set)

Return the cardinality of set. In Python 2.x, this method will raise OverflowError if the number of IPs in the set cannot be represented by Python’s Plain Integer type—that is, if the value is larger than sys.maxint. The cardinality() method will not raise this exception.

set.is_ipv6()

Return True if set is a set of IPv6 addresses, and False if it a set of IPv4 addresses. For the purposes of this method, IPv4-in-IPv6 addresses (that is, addresses in the ::ffff:0:0/96 prefix) are considered IPv6 addresses.

addr in set

Return True if addr is a member of set; False otherwise.

addr not in set

Return False if addr is a member of set; True otherwise.

set.copy()

Return a new IPSet with a copy of set.
`set.issubset(ip_iterable)`

`set <= set2`

Return `True` if every IP address in `set` is also in `set2`. Return `False` otherwise.

`set.issuperset(ip_iterable)`

`set >= set2`

Return `True` if every IP address in `set2` is also in `set`. Return `False` otherwise.

`set.union(ip_iterable[, ...])`

`set | other | ...

Return a new `IPset` containing the IP addresses in `set` and all `others`.

`set.intersection(ip_iterable[, ...])`

`set & other & ...

Return a new `IPset` containing the IP addresses common to `set` and `others`.

`set.difference(ip_iterable[, ...])`

`set - other - ...

Return a new IPset containing the IP addresses in `set` but not in `others`.

`set.symmetric_difference(ip_iterable)`

`set ^ other

Return a new IPset containing the IP addresses in either `set` or in `other` but not in both.

`set.isdisjoint(ip_iterable)`

Return `True` when none of the IP addresses in `ip_iterable` are present in `set`. Return `False` otherwise.

`set.cidr_iter()`

Return an iterator over the CIDR blocks in `set`. Each iteration returns a 2-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 set.cidr_iter()`.

`set.save(filename, compression=DEFAULT)`

Save the contents of `set` in the file `filename`. The `compression` determines the compression method used when outputting the file. Valid values are the same as those in `silk.silkfile.open()`.

The following operations and methods will modify the `IPSet`:

`set.add(addr)`

Add `addr` to `set` and return `set`. To add multiple IP addresses, use the `add_range()` or `update()` methods.

`set.discard(addr)`

Remove `addr` from `set` if `addr` is present; do nothing if it is not. Return `set`. To discard multiple IP addresses, use the `difference_update()` method. See also the `remove()` method.

`set.remove(addr)`

Similar to `discard()`, but raise `KeyError` if `addr` is not a member of `set`.  

12 SiLK-3.19.1
**set.pop()**

Remove and return an arbitrary address from set. Raise **KeyError** if set is empty.

**set.clear()**

Remove all IP addresses from set and return set.

**set.convert(version)**

Convert set to an IPv4 IPset if version is 4 or to an IPv6 IPset if version is 6. Return set. Raise **ValueError** if version is not 4 or 6. If version is 4 and set contains IPv6 addresses outside of the ::ffff:0:0/96 prefix, raise **ValueError** and leave set unchanged.

**set.add_range(start, end)**

Add all IP addresses between start and end, inclusive, to set. Raise **ValueError** if end is less than start.

**set.update(ip_iterable[, ...])**

Add the IP addresses specified in others to set; the result is the union of set and others.

**set.intersection_update(ip_iterable[, ...])**

Remove from set any IP address that does not appear in others; the result is the intersection of set and others.

**set.difference_update(ip_iterable[, ...])**

Remove from set any IP address found in others; the result is the difference of set and others.

**set.symmetric_difference_update(ip_iterable)**

Update set, keeping the IP addresses found in set or in other but not in both.

---

**RWRec Object**

An **RWRec** object represents a SiLK Flow record.

```python
class silk.RWRec([rec], [field=value], ...)
```

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

Example:

```python
>>> 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:

Accessing or setting attributes on an RWRec whose descriptions mention functions in the silk.site module causes the silk.site.init_site() function to be called with no argument if it has not yet been called successfully—that is, if silk.site.have_site_config() returns False.

rec.application
   The service port of the flow rec as set by the flow meter if the meter supports it, a 16-bit integer. The yaf(1) flow meter refers to this value as the appLabel. The default application value is 0.

rec.bytes
   The count of the number of bytes in the flow rec, a 32-bit integer. The default bytes value is 0.

rec.classname
   (READ ONLY) The class name assigned to the flow rec, a string. This value is first member of the tuple returned by the rec.classtype attribute, which see.

rec.classtype
   A 2-tuple containing the classname and the typename of the flow rec. Getting the value returns the result of ???. If that function throws an error, the result is a 2-tuple containing the string ? and a string representation of rec.classtype.id. Setting the value to (class, type) sets rec.classtype.id to the result of ???. If that function throws an error because the (class, type) pair is unknown, rec is unchanged and ValueError is thrown.

rec.classtype_id
   The ID for the class and type of the flow rec, an 8-bit integer. The default classtype_id value is 255. Changes to this value are reflected in the rec.classtype attribute. The classtype_id attribute may be set to a value that is considered invalid by the silk.site.

rec.dip
   The destination IP of the flow rec, an IPAddress object. The default dip value is IPAddress('0.0.0.0'). May be set using a string containing a valid IP address.

rec.dport
   The destination port of the flow rec, a 16-bit integer. The default dport value is 0. Since the destination port field is also used to store the values for the ICMP type and code, setting this value may modify rec.icmptype and rec.icmpcode.

rec.duration
   The duration of the flow rec, a datetime.timedelta object. The default duration value is 0. Changing the rec.duration attribute will modify the rec.etime attribute such that (rec.etime - rec.stime) == the new rec.duration. The maximum possible duration is datetime.timedelta(milliseconds=0xffffffff). See also rec.duration_secs.

rec.duration_secs
   The duration of the flow rec in seconds, a float that includes fractional seconds. The default duration_secs value is 0. Changing the rec.duration_secs attribute will modify the rec.etime attribute in the same way as changing rec.duration. The maximum possible duration_secs value is 4294967.295.

rec.etime
   The end time of the flow rec, a datetime.datetime object. The default etime value is the UNIX epoch time, datetime.datetime(1970,1,1,0,0). Changing the rec.etime attribute modifies the flow record’s duration. If the new duration would become negative or would become larger than RWRec supports, a ValueError will be raised. See also rec.etime_epoch_secs.
rec.etime_epoch_secs

The end time of the flow rec as a number of seconds since the epoch time, a float that includes fractional seconds. Epoch time is 1970-01-01 00:00:00 UTC. The default etime_epoch_secs value is 0. Changing the rec.etime_epoch_secs attribute modifies the flow record’s duration. If the new duration would become negative or would become larger than RWRec supports, a ValueError will be raised.

rec.initial_tcpflags

The TCP flags on the first packet of the flow rec, a TCPFlags object. The default initial_tcpflags value is None. The rec.initial_tcpflags attribute may be set to a new TCPFlags object, or a string or number which can be converted to a TCPFlags object by the TCPFlags() constructor. Setting rec.initial_tcpflags when rec.session_tcpflags is None sets the latter to TCPFlags("`). Setting rec.initial_tcpflags or rec.session_tcpflags sets rec.tcpflags to the binary OR of their values. Trying to set rec.initial_tcpflags when rec.protocol is not 6 (TCP) will raise an AttributeError.

rec.icmpcode

The ICMP code of the flow rec, an 8-bit integer. The default icmpcode value is 0. The value is only meaningful when rec.protocol is ICMP (1) or when rec.is_ipv6() is True and rec.protocol is ICMPv6 (58). Since a record’s ICMP type and code are stored in the destination port, setting this value may modify rec.dport.

rec.icmptype

The ICMP type of the flow rec, an 8-bit integer. The default icmptype value is 0. The value is only meaningful when rec.protocol is ICMP (1) or when rec.is_ipv6() is True and rec.protocol is ICMPv6 (58). Since a record’s ICMP type and code are stored in the destination port, setting this value may modify rec.dport.

rec.input

The SNMP interface where the flow rec entered the router or the vlanId if the packing tools are configured to capture it (see sensor.conf(5)), a 16-bit integer. The default input value is 0.

rec.nhip

The next-hop IP of the flow rec as set by the router, an IPAddress object. The default nhip value is IPAddress(’0.0.0.0’). May be set using a string containing a valid IP address.

rec.output

The SNMP interface where the flow rec exited the router or the postVlanId if the packing tools are configured to capture it (see sensor.conf(5)), a 16-bit integer. The default output value is 0.

rec.packets

The packet count for the flow rec, a 32-bit integer. The default packets value is 0.

rec.protocol

The IP protocol of the flow rec, an 8-bit integer. The default protocol value is 0. Setting rec.protocol to a value other than 6 (TCP) causes rec.initial_tcpflags and rec.session_tcpflags to be set to None.

rec.sensor

The name of the sensor where the flow rec was collected, a string. Getting the value returns the result of §???. If that function throws an error, the result is a string representation of rec.sensor_id or the string ? when sensor_id is 65535. Setting the value to sensor_name sets rec.sensor_id to the result of §???. If that function throws an error because sensor_name is unknown, rec is unchanged and ValueError is thrown.
**rec.sensor_id**

The ID of the sensor where the flow `rec` was collected, a 16-bit integer. The default sensor_id value is 65535. Changes to this value are reflected in the `rec.sensor` attribute. The sensor_id attribute may be set to a value that is considered invalid by `silk.site`.

**rec.session_tcpflags**

The union of the flags of all but the first packet in the flow `rec`, a `TCPFlags` object. The default session_tcpflags value is `None`. The `rec.session_tcpflags` attribute may be set to a new `TCPFlags` object, or a string or number which can be converted to a `TCPFlags` object by the `TCPFlags()` constructor. Setting `rec.session_tcpflags` when `rec.initial_tcpflags` is `None` sets the latter to `TCPFlags('')`. Setting `rec.initial_tcpflags` or `rec.session_tcpflags` sets `rec.tcpflags` to the binary OR of their values. Trying to set `rec.session_tcpflags` when `rec.protocol` is not 6 (TCP) will raise an `AttributeError`.

**rec.sip**

The source IP of the flow `rec`, an `IPAddr` object. The default sip value is `IPAddr('0.0.0.0')`. May be set using a string containing a valid IP address.

**rec.sport**

The source port of the flow `rec`, an integer. The default sport value is 0.

**rec.stime**

The start time of the flow `rec`, a `datetime.datetime` object. The default stime value is the UNIX epoch time, `datetime.datetime(1970,1,1,0,0)`. Modifying the `rec.stime` attribute will modify the flow’s end time such that `rec.duration` is constant. The maximum possible stime is 2038-01-19 03:14:07 UTC. See also `rec.etime_epoch_secs`.

**rec.stime_epoch_secs**

The start time of the flow `rec` as a number of seconds since the epoch time, a float that includes fractional seconds. Epoch time is 1970-01-01 00:00:00 UTC. The default stime_epoch_secs value is 0. Changing the `rec.stime_epoch_secs` attribute will modify the flow’s end time such that `rec.duration` is constant. The maximum possible stime_epoch_secs is 2147483647 (2^31-1).

**rec.tcpflags**

The union of the TCP flags of all packets in the flow `rec`, a `TCPFlags` object. The default tcpflags value is `TCPFlags('')`. The `rec.tcpflags` attribute may be set to a new `TCPFlags` object, or a string or number which can be converted to a `TCPFlags` object by the `TCPFlags()` constructor. Setting `rec.tcpflags` sets `rec.initial_tcpflags` and `rec.session_tcpflags` to `None`. Setting `rec.initial_tcpflags` or `rec.session_tcpflags` sets `rec.tcpflags` to the binary OR of their values.

**rec.timeout_killed**

Whether the flow `rec` was closed early due to timeout by the collector, a boolean. The default timeout_killed value is `False`.

**rec.timeout_started**

Whether the flow `rec` is a continuation from a timed-out flow, a boolean. The default timeout_started value is `False`.

**rec.typename**

(READ ONLY) The type name of the flow `rec`, a string. This value is second member of the tuple returned by the `rec.class_type` attribute, which see.
**rec.uniform_packets**

Whether the flow rec contained only packets of the same size, a boolean. The default uniform_packets value is False.

Supported operations and methods:

**rec.is_icmp()**

Return True if the protocol of rec is 1 (ICMP) or if the protocol of rec is 58 (ICMPv6) and rec.is_ipv6() is True. Return False otherwise.

**rec.is_ipv6()**

Return True if rec contains IPv6 addresses, False otherwise.

**rec.is_web()**

Return True if rec can be represented as a web record, False otherwise. A record can be represented as a web record if the protocol is TCP (6) and either the source or destination port is one of 80, 443, or 8080.

**rec.as_dict()**

Return a dictionary representing the contents of rec. Implicitly calls silk.site.init_site() with no arguments if silk.site.have_site_config() returns False.

**rec.to_ipv4()**

Return a new copy of rec with the IP addresses (sip, dip, and nhip) converted to IPv4. If any of these addresses cannot be converted to IPv4, (that is, if any address is not in the ::ffff:0:0/96 prefix) return None.

**rec.to_ipv6()**

Return a new copy of rec with the IP addresses (sip, dip, and nhip) converted to IPv6. Specifically, the function maps the IPv4 addresses into the ::ffff:0:0/96 prefix.

**str(rec)**

Return the string representation of rec.as_dict().

**rec1 == rec2**

Return True if rec1 is structurally equivalent to rec2. Return False otherwise.

**rec1 != rec2**

Return True if rec1 is not structurally equivalent to rec2 Return False otherwise.

**SilkFile Object**

A 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.

Creation functions:

silk.silkfile_open(filename, mode, compression=DEFAULT, notes=[], invocations=[])  
This function takes a filename, a mode, and a set of optional keyword parameters. It returns a SilkFile object. The mode should be one of the following constant values:
silk.READ
   Open file for reading
silk.WRITE
   Open file for writing
silk.APPEND
   Open file for appending

The filename should be the path to the file to open. 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 may be one of the following constants. (This list assumes SiLK was built with the required libraries. To check which compression methods are available at your site, see silk.get_configuration("COMPRESSION_METHODS").)

silk.DEFAULT
   Use the default compression scheme compiled into SiLK.

silk.NO_COMPRESSION
   Use no compression.

silk.ZLIB
   Use zlib block compression (as used by gzip(1)).

silk.LZO1X
   Use lzo1x block compression.

silk.SNAPPY
   Use snappy block compression.

If notes or invocations are set, they should be list of strings. These add annotation and invocation headers to the file. These values are visible by the rwfileinfo(1) program.

Examples:

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

silk.silkfile_fdopen(fileno, mode, filename=None, compression=DEFAULT, notes=[], invocations=[])  

This function takes an integer file descriptor, a mode, and a set of optional keyword parameters. It returns a SilkFile object. The filename parameter is used to set the value of the name attribute of the resulting object. All other parameters work as described in the silk.silkfile_open() function.

Deprecated constructor:

class silk.SilkFile(filename, mode, compression=DEFAULT, notes=[], invocations=[])  

This constructor creates a SilkFile object. The parameters are identical to those used by the silkfile_open() function. This constructor is deprecated as of SiLK 3.0.0. For future compatibility, please use the silkfile_open() function instead of the SilkFile() constructor to create SilkFile objects.
Instance attributes:

`file.name`

The filename that was used to create `file`.

`file.mode`

The mode that was used to create `file`. Valid values are `READ`, `WRITE`, or `APPEND`.

Instance methods:

`file.read()`

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

`file.write(rec)`

Write the `RWRec` `rec` to the `SilkFile` file. Return `None`.

`file.next()`

A `SilkFile` object is its own iterator. For example, `iter(file)` returns `file`. When the `SilkFile` is used as an iterator, the `next()` method is called repeatedly. This method returns the next record, or raises `StopIteration` once the end of file is reached.

`file.skip(count)`

Skip the next `count` records in `file` and return the number of records skipped. If the return value is less than `count`, the end of the file has been reached. At end of file, return 0. Since SiLK 3.19.1.

`file.notes()`

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

`file.invocations()`

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

`file.close()`

Close the file and return `None`.

**PrefixMap Object**

A `PrefixMap` object represents an immutable mapping from IP addresses or protocol/port pairs to labels. `PrefixMap` objects are created from SiLK prefix map files as created by `rwpmapbuild(1)`.

```python
class silk.PrefixMap(filename)
```

The constructor creates a prefix map initialized from the `filename`. The `PrefixMap` object will be of one of the two subtypes of `PrefixMap`: an `AddressPrefixMap` or a `ProtoPortPrefixMap`.

Supported operations and methods:
pmap[key]

Return the string label associated with key in pmap. key must be of the correct type: either an IPAddr if pmap is an AddressPrefixMap, or a 2-tuple of integers (protocol, port), if pmap is a ProtoPortPrefixMap. The method raises TypeError when the type of the key is incorrect.

pmap.get(key, default=None)

Return the string label associated with key in pmap. Return the value default if key is not in pmap, or if key is of the wrong type or value to be a key for pmap.

pmap.values()

Return a tuple of the labels defined by the PrefixMap pmap.

pmap.iterranges()

Return an iterator that will iterate over ranges of contiguous values with the same label. The return values of the iterator will be the 3-tuple (start, end, label), where start is the first element of the range, end is the last element of the range, and label is the label for that range.

Bag Object

A Bag object is a representation of a multiset. Each key represents a potential element in the set, and the key’s value represents the number of times that key is in the set. As such, it is also a reasonable representation of a mapping from keys to integers.

Please note, however, that despite its set-like properties, Bag objects are not nearly as efficient as IPSet objects when representing large contiguous ranges of key data.

In PySiLK, the Bag object is designed to look and act similar to Python dictionary objects, and in many cases Bags and dicts can be used interchangeably. There are differences, however, the primary of which is that bag[key] returns a value for all values in the key range of the bag. That value will be an integer zero for all key values that have not been incremented.

class silk.Bag(mapping=None, key_type=None, key_len=None, counter_type=None, counter_len=None)

The constructor creates a bag. All arguments are optional, and can be used as keyword arguments.

If mapping is included, the bag is initialized from that mapping. Valid mappings are:

- a Bag
- a key/value dictionary
- an iterable of key/value pairs

The key_type and key_len arguments describe the key field of the bag. The key_type should be a string from the list of valid types below. The key_len should be an integer describing the number of bytes that will represent values of key_type. The key_type argument is case-insensitive.

If key_type is not specified, it defaults to 'any-ipv6', unless silk.ipv6_enabled() is False, in which case the default is 'any-ipv4'. The one exception to this is when key_type is not specified, but key_len is specified with a value of less than 16. In this case, the default type is 'custom'.

Note: Key types that specify IPv6 addresses are not valid if silk.ipv6_enabled() returns False. An error will be thrown if they are used in this case.
If `key_len` is not specified, it defaults to the default number of bytes for the given `key_type` (which can be determined by the chart below). If specified, `key_len` must be one of the following integers: 1, 2, 4, 16.

The `counter_type` and `counter_len` arguments describe the counter value of the bag. The `counter_type` should be a string from the list of valid types below. The `counter_len` should be an integer describing the number of bytes that will represent valid of `counter_type`. The `counter_type` argument is case insensitive.

If `counter_type` is not specified, it defaults to 'custom'.

If `counter_len` is not specified, it defaults to 8. Currently, 8 is the only valid value of `counter_len`.

Here is the list of valid key and counter types, along with their default `key_len` values:

'sIPv4', 4
'dIPv4', 4
'sPort', 2
'dPort', 2
'protocol', 1
'packets', 4
'bytes', 4
'flags', 1
'sTime', 4
'duration', 4
'eTime', 4
'sensor', 2
'input', 2
'output', 2
'nhIPv4', 4
'initialFlags', 1
'sessionFlags', 1
'attributes', 1
'application', 2
'class', 1
'type', 1
'icmpTypeCode', 2
'sIPv6', 16
'dIPv6', 16
'nhIPv6', 16
Deprecation Notice: For compatibility with SiLK 2.x, the key_type argument may be a Python class. An object of the key_type class must be constructable from an integer, and it must possess an __int__() method which retrieves that integer from the object. Regardless of the maximum integer value supported by the key_type class, internally the bag will store the keys as type 'custom' with length 4.

Other constructors, all class methods:

silk.Bag.ipaddr(mapping, counter_type=None, counter_len=None)

Creates a Bag using 'any-ipv6' as the key type (or 'any-ipv4' if silk.ipv6_enabled() is False). counter_type and counter_len are used as in the standard Bag constructor. Equivalent to Bag(mapping).

silk.Bag.integer(mapping, key_len=None, counter_type=None, counter_len=None)

Creates a Bag using 'custom' as the key type (integer bag). key_len, counter_type, and counter_len are used as in the standard Bag constructor. Equivalent to Bag(mapping, key_type='custom').

silk.Bag.load(path, key_type=None)

Creates a Bag by reading a SiLK bag file. path must be a valid location of a bag. When present, the key_type argument is used as in the Bag constructor, ignoring the key type specified in the bag file. When key_type is not provided and the bag file does not contain type information, the key is set to 'custom' with a length of 4.

silk.Bag.load_ipaddr(path)

Creates an IP address bag from a SiLK bag file. Equivalent to Bag.load(path, key_type=IPv4Addr). This constructor is deprecated as of SiLK 3.2.0.

silk.Bag.load_integer(path)

Creates an integer bag from a SiLK bag file. Equivalent to Bag.load(path, key_type=int). This constructor is deprecated as of SiLK 3.2.0.

Constants:

silk.BAG_COUNTER_MAX

This constant contains the maximum possible value for Bag counters.
Other class methods:

`silk.Bag.field_types()`
Returns a tuple of strings which are valid key_type or counter_type values.

`silk.Bag.type_merge(type_a, type_b)`
Given two types from Bag.field_types(), returns the type that would be given (by default) to a bag that is a result of the co-mingling of two bags of the given types. For example: `Bag.type_merge('sport', 'dport') == 'any-port'`.

Supported operations and methods:

In the lists of operations and methods below,

- `bag` and `bag2` are Bag objects
- `key` and `key2` are IPAddrs for bags that contain IP addresses, or integers for other bags
- `value` and `value2` are integers which represent the counter associated a key in the bag
- `ipset` is an IPSet object
- `ipwildcard` is an IPWildcard object

The following operations and methods do not modify the Bag:

`bag.get_info()`
Return information about the keys and counters of the bag. The return value is a dictionary with the following keys and values:

- 'key_type'
  The current key type, as a string.
- 'key_len'
  The current key length in bytes.
- 'counter_type'
  The current counter type, as a string.
- 'counter_len'
  The current counter length in bytes.

The keys have the same names as the keyword arguments to the bag constructor. As a result, a bag with the same key and value information as an existing bag can be generated by using the following idiom: `Bag(**bag.get_info())`.

`bag.copy()`
Return a new Bag which is a copy of bag.

`bag[key]`
Return the counter value associated with key in bag.
PySiLK Reference Guide

bag[\text{key:}key2] \text{ or } bag[\text{key,}key2,...]

Return a new Bag which contains only the elements in the key range \([key, key2]\), or a new Bag containing only the given elements in the comma-separated list. In point of fact, the argument(s) in brackets can be any number of comma separated keys or key ranges. For example: bag[1,5,15:18,20] will return a bag which contains the elements 1, 5, 15, 16, 17, and 20 from bag.

bag[ipset]

Return a new Bag which contains only elements in bag that are also contained in ipset. This is only valid for IP address bags. The ipset can be included as part of a comma-separated list of slices, as above.

bag[ip wildcard]

Return a new Bag which contains only elements that are also contained in ip wildcard. This is only valid for IP address bags. The ip wildcard can be included as part of a comma-separated list of slices, as above.

key in bag

Return True if bag[key] is non-zero, False otherwise.

bag.get(key, default=None)

Return bag[key] if key is in bag, otherwise return default.

bag.items()

Return a list of \((\text{key,} \text{value})\) pairs for all keys in bag with non-zero values. This list is not guaranteed to be sorted in any order.

bag.iteritems()

Return an iterator over \((\text{key,} \text{value})\) pairs for all keys in bag with non-zero values. This iterator is not guaranteed to iterate over items in any order.

bag.sorted_iter()

Return an iterator over \((\text{key,} \text{value})\) pairs for all keys in bag with non-zero values. This iterator is guaranteed to iterate over items in key-sorted order.

bag.keys()

Return a list of keys for all keys in bag with non-zero values. This list is guaranteed to be in key-sorted order.

bag.iterkeys()

Return an iterkeys over keys for all keys in bag with non-zero values. This iterator is not guaranteed to iterate over keys in any order.

bag.values()

Return a list of values for all keys in bag with non-zero values. The list is guaranteed to be in key-sorted order.

bag.itervalues()

Return an iterator over values for all keys in bag with non-zero values. This iterator is not guaranteed to iterate over values in any order, but the order is consistent with that returned by iterkeys().
**bag.group_iterator(bag2)**

Return an iterator over keys and values of a pair of Bags. For each *key* which is in either *bag* or *bag2*, this iterator will return a (*key, value, value2*) triple, where *value* is *bag*.get(*key*), and *value2* is *bag*.get(*key*). This iterator is guaranteed to iterate over triples in *key* order.

**bag + bag2**

Add two bags together. Return a new Bag for which *newbag*[key] = *bag*[key] + *bag2*[key] for all keys in *bag* and *bag2*. Will raise an OverflowError if the resulting value for a key is greater than BAG_COUNTER_MAX. If the two bags are of different types, the resulting bag will be of a type determined by Bag.type_merge().

**bag - bag2**

Subtract two bags. Return a new Bag for which *newbag*[key] = *bag*[key] - *bag2*[key] for all keys in *bag* and *bag2*, as long as the resulting value for that key would be non-negative. If the resulting value for a key would be negative, the value of that key will be zero. If the two bags are of different types, the resulting bag will be of a type determined by Bag.type_merge().

**bag.min(bag2)**

Return a new Bag for which *newbag*[key] = min(*bag*[key], *bag2*[key]) for all keys in *bag* and *bag2*.

**bag.max(bag2)**

Return a new Bag for which *newbag*[key] = max(*bag*[key], *bag2*[key]) for all keys in *bag* and *bag2*.

**bag.div(bag2)**

Divide two bags. Return a new Bag for which *newbag*[key] = *bag*[key] / *bag2*[key]) rounded to the nearest integer for all keys in *bag* and *bag2*, as long as *bag2*[key] is non-zero. *newbag*[key] = 0 when *bag2*[key] is zero. If the two bags are of different types, the resulting bag will be of a type determined by Bag.type_merge().

**bag * integer**

**integer * bag**

Multiple a bag by a scalar. Return a new Bag for which *newbag*[key] = *bag*[key] * integer for all keys in *bag*.

**bag.intersect(set_like)**

Return a new Bag which contains *bag*[key] for each *key* where *key* in set_like is true. set_like is any argument that supports Python’s in operator, including Bags, IPSets, IPWildcards, and Python sets, lists, tuples, et cetera.

**bag.complement_intersect(set_like)**

Return a new Bag which contains *bag*[key] for each *key* where *key* in set_like is not true.

**bag.ipset()**

Return an IPSet consisting of the set of IP address key values from *bag* with non-zero values. This only works if *bag* is an IP address bag.

**bag.inversion()**

Return a new integer Bag for which all values from *bag* are inserted as key elements. Hence, if two keys in *bag* have a value of 5, *newbag*[5] will be equal to two.
bag == bag2
    Return True if the contents of bag are equivalent to the contents of bag2, False otherwise.

bag != bag2
    Return False if the contents of bag are equivalent to the contents of bag2, True otherwise.

bag.save(filename, compression=DEFAULT)
    Save the contents of bag in the file filename. The compression determines the compression method used when outputting the file. Valid values are the same as those in silk.silkfile_open().

The following operations and methods will modify the Bag:

bag.clear()
    Empty bag, such that bag[key] is zero for all keys.

bag[key] = value
    Set the number of key in bag to value.

del bag[key]
    Remove key from bag, such that bag[key] is zero.

bag.update(mapping)
    For each item in mapping, bag is modified such that for each key in mapping, the value for that key in bag will be set to the mapping’s value. Valid mappings are those accepted by the Bag() constructor.

bag.add(key[, key2[, ...]])
    Add one of each key to bag. This is the same as incrementing the value for each key by one.

bag.add(iterable)
    Add one of each key in iterable to bag. This is the same as incrementing the value for each key by one.

bag.remove(key[, key2[, ...]])
    Remove one of each key from bag. This is the same as decrementing the value for each key by one.

bag.remove(iterable)
    Remove one of each key in iterable from bag. This is the same as decrementing the value for each key by one.

bag.incr(key, value = 1)
    Increment the number of key in bag by value. value defaults to one.

bag.decr(key, value = 1)
    Decrement the number of key in bag by value. value defaults to one.

bag += bag2
    Equivalent to bag = bag + bag2, unless an OverflowError is raised, in which case bag is no longer necessarily valid. When an error is not raised, this operation takes less memory than bag = bag + bag2. This operation can change the type of bag, as determined by Bag.type_merge().

bag -= bag2
    Equivalent to bag = bag - bag2. This operation takes less memory than bag = bag - bag2. This operation can change the type of bag, as determined by Bag.type_merge().
bag *= integer

Equivalent to bag = bag * integer, unless an OverflowError is raised, in which case bag is no longer necessarily valid. When an error is not raised, this operation takes less memory than bag = bag * integer.

bag.constrain_values(min=None, max=None)

Remove key from bag if that key’s value is less than min or greater than max. At least one of min or max must be specified.

bag.constrain_keys(min=None, max=None)

Remove key from bag if that key is less than min, or greater than max. At least one of min or max must be specified.

TCPFlags Object

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

class silk.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):

flags.fin
True if the FIN flag is set on flags, False otherwise

flags.syn
True if the SYN flag is set on flags, False otherwise

flags.rst
True if the RST flag is set on flags, False otherwise

flags.psh
True if the PSH flag is set on flags, False otherwise

flags.ack
True if the ACK flag is set on flags, False otherwise

flags.urg
True if the URG flag is set on flags, False otherwise
flags.ece
    True if the ECE flag is set on flags, False otherwise

flags.cwr
    True if the CWR flag is set on flags, False otherwise

Supported operations and methods:

~flags
    Return the bitwise inversion (not) of flags

flags1 & flags2
    Return the bitwise intersection (and) of the flags from flags1 and flags2

flags1 | flags2
    Return the bitwise union (or) of the flags from flags1 and flags2.

flags1 ^ flags2
    Return the bitwise exclusive disjunction (xor) of the flags from flags1 and flags2.

int(flags)
    Return the integer value of the flags set in flags.

str(flags)
    Return a string representation of the flags set in flags.

flags.padded()
    Return a string representation of the flags set in flags. This representation will be padded with spaces such that flags will line up if printed above each other.

flags
    When used in a setting that expects a boolean, return True if any flag value is set in flags. Return False otherwise.

flags.matches(flagmask)
    Given flagmask, a string of the form high_flags/mask_flags, return True if the flags of flags match high_flags after being masked with mask_flags; False otherwise. Given a flagmask without the slash (/), return True if all bits in flagmask are set in flags. I.e., a flagmask without a slash is interpreted as "flagmask/flagmask".

Constants:
The following constants are defined:

silk.TCP_FIN
    A TCPFlags value with only the FIN flag set

silk.TCP_SYN
    A TCPFlags value with only the SYN flag set
silk.TCP_RST
A TCPFlags value with only the RST flag set

silk.TCP_PSH
A TCPFlags value with only the PSH flag set

silk.TCP_ACK
A TCPFlags value with only the ACK flag set

silk.TCP.URG
A TCPFlags value with only the URG flag set

silk.TCP.ECE
A TCPFlags value with only the ECE flag set

silk.TCP.CWR
A TCPFlags value with only the CWR flag set

FGlob Object

An FGlob object is an iterable object which iterates over filenames from a SiLK data store. It does this internally by calling the rwfglob(1) program. The FGlob object assumes that the rwfglob program is in the PATH, and will raise an exception when used if not.

Note: It is generally better to use the silk.site.repository_iter() function from the silk.site Module instead of the FGlob object, as that function does not require the external rwfglob program. However, the FGlob constructor allows you to use a different site configuration file every time, whereas the silk.site.init_site() function only supports a single site configuration file.

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

Although all arguments have defaults, at least one of classname, type, sensors, start_date must be specified. The arguments are:

classname
if given, should be a string representing the class name. If not given, defaults based on the site configuration file, silk.conf(5).

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, silk.conf.

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. The end_date cannot be
specified 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 the value in the SILK_DATA_ROOTDIR environment variable or the
compiled-in default (/data).

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

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_open(filename):
            ...

silk.site Module

The silk.site module contains functions that load the SiLK site file, and query information from that file.

silk.site.init_site(siteconf=None, rootdir=None)

Initializes the SiLK system’s site configuration. The siteconf parameter, if given, should be the path
and name of a SiLK site configuration file (see silk.conf(5)). If siteconf is omitted, the value specified
in the environment variable 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 by the rootdir argument, the directory specified in the SILK_DATA_ROOTDIR
environment variable; the data root directory that is compiled into SiLK (/data); the directories
$SILK_PATH/share/silk/ and $SILK_PATH/share/.

The rootdir parameter, if given, should be the path to a SiLK data repository that a configuration
that matches the SiLK site configuration. If rootdir is omitted, the value specified in the
SILK_DATA_ROOTDIR environment variable will be used, or if that variable is not set, the data
root directory that is compiled into SiLK (/data). The rootdir may be specified without a siteconf
argument by using rootdir as a keyword argument. I.e., init_site(rootdir=“/data”).

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 successfully once. The return value of init_site() will be
true if the site configuration was successful, or False if a site configuration file was not found. If a
siteconf parameter was specified but not found, or if a site configuration file was found but did not
parse properly, an exception will be raised instead. Once init_site() has been successfully invoked,
silk.site.have_site_config() will return True, and subsequent invocations of init_site() will raise a
RuntimeError exception.

Some silk.site methods and RWRec members require information from the silk.conf file, and
when these methods are called or members accessed, the silk.site.init_site() function is implicitly
invoked with no arguments if it has not yet been called successfully. The list of functions, methods, and attributes that exhibit this behavior include: `silk.site.sensors()`, `silk.site.classtypes()`, `silk.site.classes()`, `silk.site.types()`, `silk.site.default_types()`, `silk.site.class Sensors()`, `silk.site.classtype_id()`, `silk.site.default_class()`, `silk.site.class_sensors()`, `silk.site.sensor_id()`, `silk.site.sensor_from_id()`, `silk.site.classtype_id()`, `silk.site.class_type_from_id()`, `silk.site.set_data_rootdir()`, `silk.site.repository_iter()`, `silk.site.repository_silkfile_iter()`, `silk.site.repository_full_iter()`, `rwrec.as_dict()`, `rwrec.classname`, `rwrec.typename`, `rwrec.classtype`, and `rwrec.sensor`.

`silk.site.have_site_config()`

Return `True` if `silk.site.init_site()` has been called and was able to successfully find and load a SiLK configuration file, `False` otherwise.

`silk.site.set_data_rootdir(rootdir)`

Change the current SiLK data root directory once the `silk.conf` file has been loaded. This function can be used to change the directory used by the `silk.site` iterator functions. To change the SiLK data root directory before loading the `silk.conf` file, call `silk.site.init_site()` with a `rootdir` argument. `set_data_rootdir()` implicitly calls `silk.site.init_site()` with no arguments before changing the root directory if `silk.site.have_site_config()` returns `False`.

`silk.site.get_site_config()`

Return the current path to the SiLK site configuration file. Before `silk.site.init_site()` is called successfully, this will return the place that `init_site()` called with no arguments will first look for a configuration file. After `init_site()` has been successfully called, this will return the path to the file that `init_site()` loaded.

`silk.site.get_data_rootdir()`

Return the current SiLK data root directory.

`silk.site.sensors()`

Return a tuple of valid sensor names. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Returns an empty tuple if no site file is available.

`silk.site.classes()`

Return a tuple of valid class names. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Returns an empty tuple if no site file is available.

`silk.site.types(class)`

Return a tuple of valid type names for class `class`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `class` is not a valid class.

`silk.site.classstypes()`

Return a tuple of valid (class name, type name) tuples. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Returns an empty tuple if no site file is available.

`silk.site.default_class()`

Return the default class name. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Returns `None` if no site file is available.

`silk.site.default_types(class)`

Return a tuple of default types associated with class `class`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `class` is not a valid class.
silk.site.class_sensors(class)

Return a tuple of sensors that are in class `class`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `class` is not a valid class.

silk.site.sensor_classes(sensor)

Return a tuple of classes that are associated with `sensor`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `sensor` is not a valid sensor.

silk.site.sensor_description(sensor)

Return the sensor description as a string, or `None` if there is no description. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `sensor` is not a valid sensor.

silk.site.sensor_id(sensor)

Return the numeric sensor ID associated with the string `sensor`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `sensor` is not a valid sensor.

silk.site.sensor_from_id(id)

Return the sensor name associated with the numeric sensor ID `id`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `id` is not a valid sensor identifier.

silk.site.classstype_id( (class, type) )

Return the numeric ID associated with the tuple `(class, type)`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available, if `class` is not a valid class, or if `type` is not a valid type in `class`.

silk.site.classstype_from_id(id)

Return the `(class, type)` name pair associated with the numeric ID `id`. Implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. Throws `KeyError` if no site file is available or if `id` is not a valid identifier.

silk.site.repository_iter(start=None, end=None, classname=None, types=None, classtypes=None, sensors=None)

Return an iterator over file names in a SiLK repository. The repository is assumed to be in the data root directory that is returned by `silk.site.get_data_rootdir()` and to conform to the format of the current site configuration. This function implicitly calls `silk.site.init_site()` with no arguments if `silk.site.have_site_config()` returns `False`. See also `silk.site.repository_full_iter()` and `silk.site.repository_silkfile_iter()`.

The following types are accepted for `start` and `end`:

- a `datetime.datetime` object, which is considered to be specified to hour precision
- a `datetime.date` object, which is considered to be specified to day precision
- a string in the SiLK date format `YYYY/MM/DD[:HH]`, where the timezone depends on how SiLK was compiled; check the value of `silk.get_configuration("TIMEZONE_SUPPORT")`.

The rules for interpreting `start` and `end` are:
When both start and end are specified to hour precision, files from all hours within that time range are returned.

- When start is specified to day precision, the hour specified in end (if any) is ignored, and files for all dates between midnight at start and the end of the day represented by end are returned.
- When end is not specified and start is specified to day precision, files for that complete day are returned.
- When end is not specified and start is specified to hour precision, files for that single hour are returned.
- When neither start nor end are specified, files for the current day are returned.
- It is an error to specify end without start, or to give an end that proceeds start.

To specify classes and types, either use the classname and types parameters or use the classtypes parameter. It is an error to use classname or types when classtypes is specified.

The classname parameter should be a named class that appears in silk.site.classes(). If neither classname nor classtypes are specified, classname will default to that returned by silk.site.default_class().

The types parameter should be either a named type that appears in silk.site.types(classname) or a sequence of said named types. If neither types nor classtypes is specified, types will default to silk.site.default_types(classname).

The classtypes parameter should be a sequence of (classname, type) pairs. These pairs must be in the sequence returned by silk.site.classtypes().

The sensors parameter should be either a sensor name or a sequence of sensor names from the sequence returned by silk.site.sensors(). If sensors is left unspecified, it will default to the list of sensors supported by the given class(es).

silk.site.repository_silkfile_iter(start=None, end=None, classname=None, types=None, classtypes=None, sensors=None)

Works similarly to silk.site.repository_iter() except the file names that repository_iter() would return are opened as SilkFile objects and returned.

silk.site.repository_full_iter(start=None, end=None, classname=None, types=None, classtypes=None, sensors=None)

Works similarly to silk.site.repository_iter(). Unlike repository_iter(), this iterator’s output will include the names of files that do not exist in the repository. The iterator returns (filename, bool) pairs where the bool value represents whether the given filename exists. For more information, see the description of the --print-missing-files switch in rwfglob(1).

silk.plugin Module

silk.plugin is a module to support using PySiLK code as a plug-in to the rwfilter(1), rwcut(1), rwgroup(1), rwsort(1), rwstats(1), and rwuniq(1) applications. The module defines the following methods, which are described in the silkpython(3) manual page:

silk.plugin.register_switch(switch_name, handler=handler, [arg=needs_arg], [help=help_string])

Define the command line switch --switch_name that can be used by the PySiLK plug-in.
silk.plugin.register_filter(filter, [finalize=finalize, [initialize=initialize]])

Register the callback function filter that can be used by rwfilter to specify whether the flow record passes or fails.

silk.plugin.register_field(field_name, [add_rec_to_bin=add_rec_to_bin,]
    [bin_compare=bin_compare,] [bin_bytes=bin_bytes,] [bin_merge=bin_merge,]
    [bin_to_text=bin_to_text,] [column_width=column_width,] [description=description,]
    [initial_value=initial_value,] [initialize=initialize,] [rec_to_bin=rec_to_bin,]
    [rec_to_text=rec_to_text])

Define the new key field or aggregate value field named field_name. Key fields can be used in rwcut, rwgroup, rwsort, rwstats, and rwuniq. Aggregate value fields can be used in rwstats and rwuniq. Creating a field requires specifying one or more callback functions---the functions required depend on the application(s) where the field will be used. To simplify field creation for common field types, the remaining functions can be used instead.

silk.plugin.register_int_field(field_name, int_function, min, max, [width])

Create the key field field_name whose value is an unsigned integer.

silk.plugin.register_ipv4_field(field_name, ipv4_function, [width])

Create the key field field_name whose value is an IPv4 address.

silk.plugin.register_ip_field(field_name, ipv4_function, [width])

Create the key field field_name whose value is an IPv4 or IPv6 address.

silk.plugin.register_enum_field(field_name, enum_function, width, [ordering])

Create the key field field_name whose value is a Python object (often a string).

silk.plugin.register_int_sum_aggregator(agg_value_name, int_function, [max_sum], [width])

Create the aggregate value field agg_value_name that maintains a running sum as an unsigned integer.

silk.plugin.register_int_max_aggregator(agg_value_name, int_function, [max_max], [width])

Create the aggregate value field agg_value_name that maintains the maximum unsigned integer value.

silk.plugin.register_int_min_aggregator(agg_value_name, int_function, [max_min], [width])

Create the aggregate value field agg_value_name that maintains the minimum unsigned integer value.

EXAMPLE

Using PySiLK

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.

The code reads each record in a SiLK flow file, checks whether the record’s source port is 80/tcp or 8080/tcp and its volume is larger than 3 packets and 120 bytes, stores the destination IP of matching records in an IPset, and writes the IPset to a destination file. In addition, it prints the number of unique destination addresses and the addresses themselves to the standard output. Additional explanations can be found in-line in the comments.

```python
#!/usr/bin/python
```
# Use print functions (Compatible with Python 3.0; Requires 2.6+)
from __future__ import print_function #Python2.6 or later required

# 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])
        sys.exit(1)

    # Open a silk flow file for reading
    infile = silkfile_open(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 values
        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
    try:
        destset.save(sys.argv[2])
    except:
        sys.exit("Unable to write to %s" % 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()

Adjusting the Class and Type Fields of a Flow File

Normally SiLK flow records get stamped with a class as flow records are recorded in the repository. However, if you are importing raw packet data or need to change some records that inadvertently have the wrong class/type, PySiLK makes it easy to fix.

The example below sets the class to "all" and assigns a type of "in", "inweb", "out", or "outweb" to each record in an input file. The direction (in or out) is defined by an IPset that represents the internal network (traffic that neither comes from nor goes to the internal network is discarded in this example). Web/non-web flows are separated based on port.

```python
#!/usr/bin/python

from __future__ import print_function #Python2.6 or later required
from silk import *
import silk.site
import sys # for command line args
from datetime import timedelta # for date math

webports = (80,443,8080)
inwebtype = ("all","inweb")
inintype = ("all","in")
outwebtype = ("all","outweb")
outtype = ("all","out")

def main():
    if len(sys.argv) != 4:
        print("Usage: %s infile setfile outfile" % sys.argv[0])
        sys.exit(1)

    # open the SiLK file for reading
    infile = silkfile_open(sys.argv[1], READ)

    # open the set file which represents my internal network
    #print(sys.argv[2])
    setfile = IPSet.load (sys.argv[2])

    # open the modified output file
    outfile = silkfile.open(sys.argv[3], WRITE)

    # loop over the records in the file, shift time and write the update:
    for rec in infile:
        #
        # If the src ip is in the set, it's going out.
        # If the dst ip is in the set, it's coming in.
```
# If neither IP is in the set, discard the record.
if (rec.sport in webports) or (rec.dport in webports):
    if rec.sip in setfile:
        rec.classtype = outwebtype
        outfile.write(rec)
    elif rec.dip in setfile:
        rec.classtype = inwebtype
        outfile.write(rec)
else:
    if rec.sip in setfile:
        rec.classtype = outtype
        outfile.write(rec)
    elif rec.dip in setfile:
        rec.classtype = intype
        outfile.write(rec)

# clean up
outfile.close()
infile.close()

if __name__ == '__main__':
    main()

Changing Timestamps in a Flow File

On occasion you may find that you need to adjust all the timestamps for a SiLK flow file. For example, the flow file came from a packet capture file that was collected in a different time zone and had to be shifted a number of hours. Another possibility is if you need to adjust files because you determine the clock time was off.

It is relatively simple to change the timestamps using PySiLK. The sample code for changing data to another time zone is shown below; a minor change would shift the data by seconds instead of hours.

#!/usr/bin/python

from __future__ import print_function #Python2.6 or later required
from silk import *
import sys # for command line args
from datetime import timedelta # for date math

def main():
    if len(sys.argv) != 4:
        print ("Usage: %s infile offset-hours outfile" % sys.argv[0])
        sys.exit(1)

    # open the SiLK file for reading
    infile = silkfile_open(sys.argv[1], READ)

    # create the time offset object
    offset = timedelta(hours=int(sys.argv[2]))

    # read the file and update timestamps
    for rec in infile:
        rec.ts = rec.ts + offset
        outfile.write(rec)

    outfile.close()
# open the modified output file
outfile = silkfile_open(sys.argv[3], WRITE)

# loop over the records in the file, shift time and write the update:
for rec in infile:
    rec.stime = rec.stime + offset
    outfile.write(rec)

# clean up
outfile.close()
infile.close()

if __name__ == '__main__':
    main()

Grouping FTP Flow Records

The following script attempts to group all flows representing one direction of an FTP session and print them together. It takes as an argument the name of a file containing raw SiLK records sorted by start time and port number (`rwsort --fields=stime,sport`). The script extracts from the file all flows that potentially represent FTP traffic. We define a possible FTP flow as any flow where:

- the source port is 21 (FTP control channel)
- the source port is 20 (FTP data transfer port)
- both the source port and destination port are ephemeral (data transfer)

If a flow record has a source port of 21, the script adds the source and destination address to the list of possible FTP groups. The script categorizes each data transfer flow (source port 20 or ephemeral to ephemeral) according to its source and destination IP address pair. If a flow from the control channel with the same source and destination IP address exists the source and destination ports in the flow are added to the list of ports associated with the control channel interaction, otherwise the script lists the data transfer as being unclassified. After the entire file is processed, all FTP sessions that have been grouped are displayed.

#!/usr/bin/python

from __future__ import print_function #Python2.6 or later required
# import the necessary modules
import silk
import sys

# Test that the argument number is correct
if (len(sys.argv) != 2):
    print("Must supply a SiLK data file.")
    sys.exit()

# open the SiLK file for reading
rawFile=silk.silkfile_open(sys.argv[1], silk.READ)
# Initialize the record structure
# Unclassified will be the record ephemeral to ephemeral
# connections that don’t appear to have a control channel
interactions = {
    "Unclassified": []
}

# Count of records processed
count = 0

# Process the input file
for rec in rawFile:
    count += 1
    key = "%15s <--> %15s"%(rec.sip, rec.dip)
    if (rec.sport == 21):
        if not key in interactions:
            interactions[key] = []
        else:
            interactions[key].append("%5d <--> %5d"%(rec.sport, rec.dport))
    else:
        interactions["Unclassified"].append("%15s:%5d <--> %15s:%5d"%(rec.sip, rec.sport, rec.dip, rec.dport))

# Print the count of all records
print(str(count) + " records processed")

# Print the groups of FTP flows
keyList = sorted(interactions.keys())
for key in keyList:
    print("\n" + key + " " + str(len(interactions[key])))
    if (key != "Unclassified"):
        for line in interactions[key]:
            print(" " + line)

Example output of the script:

184 records processed

xxx.xxx.xxx.236 <--> yyy.yyy.yyy.231 3
  20 <--> 56180
  20 <--> 56180
  20 <--> 58354

Unclassified 158

ENVIRONMENT

The following environment variables affect the tools in the SiLK tool suite.
SILK_CONFIG_FILE

This environment variable contains the location of the site configuration file, silk.conf. This variable will be used by silk.site.init_site() if no argument is passed to that method.

SILK_DATA_ROOTDIR

This variable gives the root of directory tree where the data store of SiLK Flow files is maintained, overriding the location that is compiled into the tools (/data). This variable will be used by the FGlob constructor unless an explicit data_roottdir value is specified. In addition, the silk.site.init_site() may search for the site configuration file, silk.conf, in this directory.

SILK_COUNTRY_CODES

This environment variable gives the location of the country code mapping file that the silk.init_country_codes() function will use when no name is given to that function. The value of this environment variable may be a complete path or a file relative to the SILK_PATH. See the FILES section for standard locations of this file.

SILK_CLOBBER

The SiLK tools normally refuse to overwrite existing files. Setting SILK_CLOBBER to a non-empty value removes this restriction.

SILK_PATH

This environment variable gives the root of the install tree. When searching for configuration files, PySiLK may use this environment variable. See the FILES section for details.

PYTHONPATH

This is the search path that Python uses to find modules and extensions. The SiLK Python extension described in this document may be installed outside Python’s installation tree; for example, in SiLK’s installation tree. It may be necessary to set or modify the PYTHONPATH environment variable so Python can find the SiLK extension.

PYTHONVERBOSE

If the SiLK Python extension fails to load, setting this environment variable to a non-empty string may help you debug the issue.

SILK_PYTHON_TRACEBACK

When set, Python plug-ins (see silkpython(3)) will output trace back information regarding Python errors to the standard error.

PATH

This is the standard search path for executable programs. The FGlob constructor will invoke the rwfglob(1) program; the directory containing rwfglob should be included in the PATH.

TZ

When a SiLK installation is built to use the local timezone (to determine if this is the case, check the value of silk.get_configuration("TIMEZONE_SUPPORT")), the value of the TZ environment variable determines the timezone in which silk.site.repository_iter() parses timestamp strings. If the TZ environment variable is not set, the default timezone is used. Setting TZ to 0 or the empty string causes timestamps to be parsed as UTC. The value of the TZ environment variable is ignored when the SiLK installation uses utc. For system information on the TZ variable, see tzset(3).
FILES

${SILK\_CONFIG\_FILE}$

ROOT\_DIRECTORY/silk.conf

${SILK\_PATH}$/share/silk/silk.conf

${SILK\_PATH}$/share/silk.conf

/usr/local/share/silk/silk.conf

/usr/local/share/silk.conf

Possible locations for the SiLK site configuration file which are checked when no argument is passed to \texttt{silk\_site\_init\_site()}.  

${SILK\_COUNTRY\_CODES}$

${SILK\_PATH}$/share/silk/country\_codes.pmap

${SILK\_PATH}$/share/silk/country\_codes.pmap

/usr/local/share/silk/country\_codes.pmap

/usr/local/share/country\_codes.pmap

Possible locations for the country code mapping file used by \texttt{silk\_init\_country\_codes()} when no name is given to the function.

${SILK\_DATA\_ROOTDIR}$/

/data/

Locations for the root directory of the data repository. The \texttt{silk\_site\_init\_site()} may search for the site configuration file, \texttt{silk\_conf}, in this directory.

SEE ALSO

silkpython

SiLK Python plug-in

SYNOPSIS

rwfilter --python-file=FILENAME [--python-file=FILENAME ...] ... 

rwfilter --python-expr=PYTHON_EXPRESSION ...

rwcut --python-file=FILENAME [--python-file=FILENAME ...]
     --fields=FIELDS ...

rwgroup --python-file=FILENAME [--python-file=FILENAME ...]
     --id-fields=FIELDS ...

rwsort --python-file=FILENAME [--python-file=FILENAME ...]
     --fields=FIELDS ...

rwstats --python-file=FILENAME [--python-file=FILENAME ...]
     --fields=FIELDS --values=VALUES ...

rwuniq --python-file=FILENAME [--python-file=FILENAME ...]
     --fields=FIELDS --values=VALUES ...

DESCRIPTION

The SiLK Python plug-in provides a way to use PySiLK (the SiLK extension for python(1) described in pysilk(3)) to extend the capability of several SiLK tools.

- In rwfilter(1), new partitioning rules can be defined in PySiLK to determine whether a SiLK Flow record is written to the --pass-destination or --fail-destination.
- In rwcut(1), new fields can be defined in PySiLK and displayed for each record.
New fields can also be defined in `rwgroup(1)` and `rwsort(1)`. These fields are used as part of the key when grouping or sorting the records.

For `rwstats(1)` and `rwuniq(1)`, two types of fields can be defined: `Key fields` are used to categorize the SiLK Flow records into bins, and `aggregate value fields` compute a value across all the SiLK Flow records that are categorized into a bin. (An example of a built-in aggregate value field is the number of packets that were seen for all flow records that match a particular key.)

To extend the SiLK tools using PySiLK, the user writes a Python file that calls Python functions defined in the `silk.plugin` Python module and described in this manual page. When the user specifies the `--python-file` switch to a SiLK application, the application loads the Python file and makes the new functionality available.

The following sections will describe

- how to create a command line switch with PySiLK that allows one to modify the run-time behavior of their PySiLK code
- how to use PySiLK with `rwfilter`
- a simple API for creating fields in `rwcut`, `rwgroup`, `rwsort`, `rwstats`, and `rwuniq`
- the advanced API for creating fields in those applications

Typically you will not need to explicitly import the `silk.plugin` module, since the `--python-file` switch does this for you. In a module used by a Python plug-in, the module can gain access to the functions defined in this manual page by importing them from `silk.plugin`:

```python
from silk.plugin import *
```

**Hint:** If you want to check whether the Python code in `FILENAME` is defining the switches and fields you expect, you can load the Python file and examine the output of `--help`, for example:

```
rwcut --python-file=FILENAME --help
```

**User-defined command line switches**

Command line switches can be added and handled from within a SiLK Python plug-in. In order to add a new switch, use the following function:

```
register_switch(switch_name, handler=handler_func, [arg=needs_arg], [help=help_string])
```

**switch_name**

Provides the name of the switch you are registering, a string. Do not include the leading `--` in the name. If a switch already exists with the name `switch_name`, the application will exit with an error message.

**handler_func**

```
handler_func([string]). Names a function that will be called by the application while it is processing its command line if and only if the command line includes the switch `--switch_name`. (If the switch is not given, the `handler_func` function will not be called.) When the `arg` parameter is specified
```
and its value is `False`, the `handler_func` function will be called with no arguments. Otherwise, the `handler_func` function will be called with a single argument: a string representing the value the user passed to the `--switch_name` switch. The return value from this function is ignored. Note that the `register_switch()` function requires a `handler` argument which must be passed by keyword.

`needs_arg`

Specifies a boolean value that determines whether the user must specify an argument to `--switch_name`, and determines whether the `handler_func` function should expect an argument. When `arg` is not specified or `needs_arg` is `True`, the user must specify an argument to `--switch_name` and the `handler_func` function will be called with a single argument. When `needs_arg` is `False`, it is an error to specify an argument to `--switch_name` and `handler_func` will be called with no arguments.

`help_string`

Provides the usage text to print describing this switch when the user runs the application with the `--help` switch. This argument is optional; when it is not provided, a simple "No help for this switch" message is printed.

**rwfilter usage**

When used in conjunction with `rwfilter(1)`, the SiLK Python plug-in allows users to define arbitrary partitioning criteria using the SiLK extension to the Python programming language. To use this capability, the user creates a Python file and specifies its name with the `--python-file` switch in `rwfilter`. The file should call the `register_filter()` function for each filter that it wants to create:

```python
register_filter(filter_func, [finalize=finalize_func], [initialize=initialize_func])
```

`filter_func`

```python
Boolean = filter_func(silk.RWRec). Names a function that must accept a single argument, a silk.RWRec object (see `pysilk(3)`). When the `rwfilter` program is run, it finds the records that match the selection options, and hands each record to the built-in partitioning switches. A record that passes all of the built-in switches is handed to the first Python `filter_func()` function as an RWRec object. The return value of the function determines what happens to the record. The record fails the `filter_func()` function (and the record is immediately written to the `--fail-destination`, if specified) when the function returns one of the following: `False`, `None`, numeric zero of any type, an empty string, or an empty container (including strings, tuples, lists, dictionaries, sets, and frozensets). If the function returns any other value, the record passes the first `filter_func()` function, and the record is handed to the next Python `filter_func()` function. If all `filter_func()` functions pass the record, the record is written to the `--pass-destination`, if specified. (Note that when the `--plugin` switch is present, the code it specifies will be called after the PySiLK code.)
```

`initialize_func`

```python
initialize_func(). Names a function that takes no arguments. When this function is specified, is will be called after `rwfilter` has completed its argument processing, and just before `rwfilter` opens the first input file. The return value of this function is ignored.
```

`finalize_func`

```python
finalize_func(). Names a function that takes no arguments. When this function is specified, it will be called after all flow records have been processed. One use of the these functions is to print any statistics that the `filter_func()` function was computing. The return value from this function is ignored.
```
If `register_filter()` is called multiple times, the `filter_func()`, `initialize_func()`, and `finalize_func()` functions will be invoked in the order in which the `register_filter()` functions were seen.

**NOTE:** For backwards compatibility, when the file named by `--python-file` does not call `register_filter()`, `rwfilter` will search the Python file for functions named `rwfilter()` and `finalize()`. If it finds the `rwfilter()` function, `rwfilter` will act as if the file contained:

```
register_filter(rwfilter, finalize=finalize)
```

The `--python-file` switch requires the user to create a file containing Python code. To allow the user to write a small filtering check in Python, `rwfilter` supports the `--python-expr` switch. The value of the switch should be a Python expression whose result determines whether a given record passes or fails, using the same criterion as the `filter_func()` function described above. In the expression, the variable `rec` is bound to the current `silk.RWRec` object. There is no support for the `initialize_func()` and `finalize_func()` functions. The user may consider `--python-expr=PYTHON_EXPRESSION` as being implemented by

```python
from silk import *

def temp_filter(rec):
    return (PYTHON_EXPRESSION)

register_filter(temp_filter)
```

The `--python-file` and `--python-expr` switches allow for much flexibility but at the cost of speed: converting a SiLK Flow record into an `RWRec` is expensive relative to most operations in `rwfilter`. The user should use `rwfilter`'s built-in partitioning switches to whittle down the input as much as possible, and only use the Python code to do what is difficult or impossible to do otherwise.

### Simple field registration functions

The `silk.plugin` module defines a function that can be used to define fields for use in `rwcut`, `rwgroup`, `rwsort`, `rwstats`, and `rwuniq`. That function is powerful, but it is also complex. To make it easy to define fields for the common cases, the `silk.plugin` provides the functions described in this section that create a key field or an aggregate value field. The advanced function is described later in this manual page (Advanced field registration function).

Once you have created a key field or aggregate value field, you must include the field’s name in the argument to the `--fields` or `--values` switch to tell the application to use the field.

#### Integer key field

The following function is used to create a key field whose value is an unsigned integer.

```
register_int_field(field_name, int_function, min, max, [width])
```

- **field_name**
  The name of the new field, a string. If you attempt to add a key field that already exists, you will get an an error message.

- **int_function**
  `int = int_function(silk.RWRec)`. A function that accepts a `silk.RWRec` object as its sole argument, and returns an unsigned integer which represents the value of this field for the given record.
min

A number representing the minimum integer value for the field. If \texttt{int\_function} returns a value less than \texttt{min}, an error is raised.

max

A number representing the maximum integer value for the field. If \texttt{int\_function} returns a value greater than \texttt{max}, an error is raised.

width

The column width to use when displaying the field. This parameter is optional; the default is the number of digits necessary to display the integer \texttt{max}.

IPv4 address key field

This function is used to create a key field whose value is an IPv4 address. (See also \texttt{register\_ip\_field()}).

\begin{verbatim}
register_ipv4_field(field_name, ipv4_function, [width])
\end{verbatim}

\texttt{field\_name}

The name of the new field, a string. If you attempt to add a key field that already exists, you will get an an error message.

\texttt{ipv4\_function}

\begin{verbatim}
silk.IPv4Addr = ipv4_function(silk.RWRec)
\end{verbatim}

A function that accepts a \texttt{silk.RWRec} object as its sole argument, and returns a \texttt{silk.IPv4Addr} object. This \texttt{IPv4Addr} object will be the IPv4 address that represents the value of this field for the given record.

width

The column width to use when displaying the field. This parameter is optional, and it defaults to 15.

IP address key field

The next function is used to create a key field whose value is an IPv4 or IPv6 address.

\begin{verbatim}
register_ip_field(field_name, ip_function, [width])
\end{verbatim}

\texttt{field\_name}

The name of the new field, a string. If you attempt to add a key field that already exists, you will get an an error message.

\texttt{ip\_function}

\begin{verbatim}
silk.IPAddr = ip_function(silk.RWRec)
\end{verbatim}

A function that accepts a \texttt{silk.RWRec} object as its sole argument, and returns a \texttt{silk.IPAddr} object which represents the value of this field for the given record.

width

The column width to use when displaying the field. This parameter is optional. The default width is 39.
This key field requires more memory internally than fields registered by the `register_ipv4_field()` function. If SiLK is compiled without IPv6 support, `register_ipv4_field()` works exactly like `register_ip_field()`, including the default width of 15.

**Enumerated object key field**

The following function is used to create a key field whose value is any Python object. The maximum number of different objects that can be represented is 4,294,967,296, or $2^{32}$.

\[
\text{register_enum_field}(\text{field\_name}, \text{enum\_function}, \text{width}, [\text{ordering}])
\]

**field\_name**

The name of the new field, a string. If you attempt to add a key field that already exists, you will get an an error message.

**enum\_function**

\[
\text{object} = \text{enum\_function}(\text{silk.RWRec})
\]

A function that accepts a `silk.RWRec` object as its sole argument, and returns a Python object which represents the value of this field for the given record. For typical usage, the Python objects returned by the `enum\_function` will be strings representing some categorical value.

**width**

The column width to use when displaying this field. The parameter is required.

**ordering**

A list of objects used to determine ordering for `rwsort` and `rwuniq`. This parameter is optional. If specified, it lists the objects in the order in which they should be sorted. If the `enum\_function` returns a object that is not in `ordering`, the object will be sorted after all the objects in `ordering`.

**Integer sum aggregate value field**

This function is used to create an aggregate value field that maintains a running unsigned integer sum.

\[
\text{register_int\_sum\_aggregator}(\text{agg\_value\_name}, \text{int\_function}, [\text{max\_sum}], [\text{width}])
\]

**agg\_value\_name**

The name of the new aggregate value field, a string. The `agg\_value\_name` must be unique among all aggregate values, but an aggregate value field and key field can have the same name.

**int\_function**

\[
\text{int} = \text{int\_function}(\text{silk.RWRec})
\]

A function that accepts a `silk.RWRec` object as its sole argument, and returns an unsigned integer which represents the value that should be added to the running sum for the current bin.

**max\_sum**

The maximum possible sum. This parameter is optional; if not specified, the default is $2^{64}-1$ ($18,446,744,073,709,551,615$).

**width**

The column width to use when displaying the aggregate value. This parameter is optional. The default is the number of digits necessary to display `max\_sum`. 
Integer maximum aggregate value field

The following function is used to create an aggregate value field that maintains the maximum unsigned integer value.

```
register_int_max_aggregator(agg_value_name, int_function, [max], [width])
```

**agg_value_name**

The name of the new aggregate value field, a string. The `agg_value_name` must be unique among all aggregate values, but an aggregate value field and key field can have the same name.

**int_function**

```
int = int_function(silk.RWRec)
```

A function that accepts a `silk.RWRec` object as its sole argument, and returns an integer which represents the value that should be considered for the current highest value for the current bin.

**max**

The maximum possible value for the maximum. This parameter is optional; if not specified, the default is $2^{64}-1$ (18,446,744,073,709,551,615).

**width**

The column width to use when displaying the aggregate value. This parameter is optional. The default is the number of digits necessary to display `max`.

Integer minimum aggregate value field

This function is used to create an aggregate value field that maintains the minimum unsigned integer value.

```
register_int_min_aggregator(agg_value_name, int_function, [max], [width])
```

**agg_value_name**

The name of the new aggregate value field, a string. The `agg_value_name` must be unique among all aggregate values, but an aggregate value field and key field can have the same name.

**int_function**

```
int = int_function(silk.RWRec)
```

A function that accepts a `silk.RWRec` object as its sole argument, and returns an integer which represents the value that should be considered for the current lowest value for the current bin.

**max**

The maximum possible value for the minimum. When this optional parameter is not specified, the default is $2^{64}-1$ (18,446,744,073,709,551,615).

**width**

The column width to use when displaying the aggregate value. This parameter is optional. The default is the number of digits necessary to display `max`.
Advanced field registration function

The previous section provided functions to register a key field or an aggregate value field when dealing with common objects. When you need to use a complex object, or you want more control over how the object is handled in PySiLK, you can use the `register_field()` function described in this section.

Many of the arguments to the `register_field()` function are callback functions that you must create and that the application will invoke. (The simple registration functions above have already taken care of defining these callback functions.)

Often the callback functions for handling fields will either take (as a parameter) or return a representation of a numeric value that can be processed from C. The most efficient way to handle these representations is as a string containing binary characters, including the null byte. We will use the term “byte sequence” for these representations; other possible terms include “array of bytes”, “byte strings”, or ”binary values”. For hints on creating byte sequences from Python, see the Byte sequences section below.

To define a new field or aggregate value, the user calls:

```python
register_field(field_name, [add_rec_to_bin=add_rec_to_bin_func], [bin_compare=bin_compare_func], [bin_bytes=bin_bytes_value], [bin_merge=bin_merge_func], [bin_to_text=bin_to_text_func], [column_width=column_width_value], [description=description_string], [initial_value=initial_value], [initialize=initialize_func], [rec_to_bin=rec_to_bin_func], [rec_to_text=rec_to_text_func])
```

Although the keyword arguments to `register_field()` are all optional from Python’s perspective, certain keyword arguments must be present before an application will define the key or aggregate value. The following table summarizes the keyword arguments used by each application. An F means the argument is required for a key field, an A means the argument is required for an aggregate value field, f and a mean the application will use the argument for a key field or an aggregate value if the argument is present, and a dot means the application completely ignores the argument.

<table>
<thead>
<tr>
<th></th>
<th>rwcut</th>
<th>rwgroup</th>
<th>rwsort</th>
<th>rwstats</th>
<th>rwuniq</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_rec_to_bin</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>bin_compare</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>bin_bytes</td>
<td>F</td>
<td>F</td>
<td>F,A</td>
<td>F,A</td>
<td></td>
</tr>
<tr>
<td>bin_merge</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>bin_to_text</td>
<td>.</td>
<td>.</td>
<td>F,A</td>
<td>F,A</td>
<td></td>
</tr>
<tr>
<td>column_width</td>
<td>F</td>
<td>.</td>
<td>F,A</td>
<td>F,A</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>f</td>
<td>f</td>
<td>f,a</td>
<td>f,a</td>
<td></td>
</tr>
<tr>
<td>initialize</td>
<td>.</td>
<td>.</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>rec_to_bin</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>rec_to_text</td>
<td>F</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

The following sections describe how to use `register_field()` in each application.

rwcut usage

The purpose of `rwcut(1)` is to print attributes of (or attributes derived from) every SiLK record it reads as input. A plug-in used by `rwcut` must produce a printable (textual) attribute from a SiLK record. To define a new attribute, the `register_field()` method should be called as shown:

```python
register_field(field_name, column_width=column_width_value, rec_to_text=rec_to_text_func, [description=description_string], [initialize=initialize_func])
```
field_name

Names the field being defined, a string. If you attempt to add a field that already exists, you will get an an error message. To display the field, include field_name in the argument to the --fields switch.

column_width_value

Specifies the length of the longest printable representation. rwcut will use it as the width for the field_name column when columnar output is selected.

rec_to_text_func

string = rec_to_text_func(silk.RWRec). Names a callback function that takes a silk.RWRec object as its sole argument and produces a printable representation of the field being defined. The length of the returned text should not be greater than column_width_value. If the value returned from this function is not a string, the returned value is converted to a string by the Python str() function.

description_string

Provides a string giving a brief description of the field, suitable for printing in --help-fields output. This argument is optional.

initialize_func

initialize_func(). Names a callback function that will be invoked after the application has completed its argument processing, and just before it opens the first input file. This function is only called when --fields includes field_name. The function takes no arguments and its return value is ignored. This argument is optional.

If the rec_to_text argument is not present, the register_field() function will do nothing when called from rwcut. If the column_width argument is missing, rwcut will complain that the textual width of the plug-in field is 0.

rwgroup and rwsort usage

The rwsort(1) tool sorts SiLK records by their attributes or attributes derived from them. rwgroup(1) reads sorted SiLK records and writes a common value into the next hop IP field of all records that have common attributes. The output from both of these tools is a stream of SiLK records (the output typically includes every record that was read as input). A plug-in used by these tools must return a value that the application can use internally to compare records. To define a new field that may be included in the --id-fields switch to rwgroup or the --fields switch to rwsort, the register_field() method should be invoked as follows:

register_field(field_name, bin_bytes=bin_bytes_value, rec_to_bin=rec_to_bin_func, [description=description_string], [initialize=initialize_func])

field_name

Names the field being defined, a string. If you attempt to add a field that already exists, you will get an an error message. To have rwgroup or rwsort use this field, include field_name in the argument to --id-fields or --fields.

bin_bytes_value

Specifies a positive integer giving the length, in bytes, of the byte sequence that the rec_to_bin_func() function produces; the byte sequence must be exactly this length.
**rec_to_bin_func**

`byte-sequence = rec_to_bin_func(silk.RWRec)`. Names a callback function that takes a `silk.RWRec` object and returns a byte sequence that represents the field being defined. The returned value should be exactly `bin_bytes_value` bytes long. For proper grouping or sorting, the byte sequence should be returned in network byte order (i.e., big endian).

**description_string**

Provides a string giving a brief description of the field, suitable for printing in `--help-fields` output. This argument is optional.

**initialize_func**

`initialize_func()`. Names a callback function that will be invoked after the application has completed its argument processing, and just before it opens the first input file. This function is only called when `field_name` is included in the list of fields. The function takes no arguments and its return value is ignored. This argument is optional.

If the `rec_to_bin` argument is not present, the `register_field()` function will do nothing when called from `rwgroup` or `rwsort`. If the `bin_bytes` argument is missing, `rwgroup` or `rwsort` will complain that the binary width of the plug-in field is 0.

**rwstats and rwuniq usage**

`rwstats(1)` and `rwuniq(1)` group SiLK records into bins based on key fields. Once a record is matched to a bin, the record is used to update the aggregate values (e.g., the sum of bytes) that are being computed, and the record is discarded. Once all records have been processed, the key fields and the aggregate values are printed.

**Key Field**

A plug-in used by `rwstats` or `rwuniq` for creating a new key field must return a value that the application can use internally to compare records, and there must be a function that converts that value to a printable representation. The following invocation of `register_field()` will produce a key field that can be used in the `--fields` switch of `rwstats` or `rwuniq`:

```python
register_field(field_name, bin_bytes=bin_bytes_value, bin_to_text=bin_to_text_func, column_width=column_width_value, rec_to_bin=rec_to_bin_func, [description=description_string], [initialize=initialize_func])
```

The arguments are:

**field_name**

Contains the name of the field being defined, a string. If you attempt to add a field that already exists, you will get an an error message. The field will only be active when `field_name` is specified as an argument to `--fields`.

**bin_bytes_value**

Contains a positive integer giving the length, in bytes, of the byte sequence that the `rec_to_bin_func()` function produces and that the `bin_to_text_func()` function accepts. The byte sequences must be exactly this length.

**bin_to_text_func**
string = bin_to_text_func(byte-sequence). Names a callback function that takes a byte sequence, of length bin_bytes_value, as produced by the rec_to_bin_func() function and returns a printable representation of the byte sequence. The length of the text should be no longer than the value specified by column_width. If the value returned from this function is not a string, the returned value is converted to a string by the Python str() function.

column_width_value

Contains a positive integer specifying the length of the longest textual field that the bin_to_text_func() callback function returns. This length will used as the column width when columnar output is requested.

rec_to_bin_func

byte-sequence = rec_to_bin_func(silk.RWRec). Names a callback function that takes a silk.RWRec object and returns a byte sequence that represents the field being defined. The returned value should be exactly bin_bytes_value bytes long. For proper sorting, the byte sequence should be returned in network byte order (i.e., big endian).

description_string

Provides a string giving a brief description of the field, suitable for printing in --help-fields output. This argument is optional.

initialize_func

initialize_func(). Names a callback function that is called after the command line arguments have been processed, and before opening the first file. This function is only called when --fields includes field_name. The function takes no arguments and its return value is ignored. This argument is optional.

Aggregate Value

A plug-in used by rwstats or rwuniq for creating a new aggregate value must be able to use a SiLK record to update an aggregate value, take two aggregate values and merge them to a new value, and convert that aggregate value to a printable representation. To use an aggregate value for ordering the bins in rwstats, the plug-in must also define a function to compare two aggregate values. The aggregate values are represented as byte sequences.

To define a new aggregate value in rwstats, the user calls:

register_field(agg_value_name, add_rec_to_bin=add_rec_to_bin_func, bin_bytes=bin_bytes_value, bin_merge=bin_merge_func, bin_to_text=bin_to_text_func, column_width=column_width_value, [bin_compare=bin_compare_func], [description=description_string], [initial_value=initial_value], [initialize=initialize_func])

The call to define a new aggregate value in rwuniq is nearly identical:

register_field(agg_value_name, add_rec_to_bin=add_rec_to_bin_func, bin_bytes=bin_bytes_value, bin_merge=bin_merge_func, bin_to_text=bin_to_text_func, column_width=column_width_value, [description=description_string], [initial_value=initial_value], [initialize=initialize_func])

The arguments are:

agg_value_name

Contains the name of the aggregate value field being defined, a string. The name of value must be unique among all aggregate values, but an aggregate value field and key field can have the same name.
The value will only be active when agg_value_name is specified as an argument to --values.
add_rec_to_bin_func

\[
\text{byte-sequence} = \text{add_rec_to_bin_func}(\text{silk.RWRec}, \text{byte-sequence})
\]

Names a callback function whose two arguments are a silk.RWRec object and an aggregate value. The function updates the aggregate value with data from the record and returns a new aggregate value. Both aggregate values are represented as byte sequences of exactly \text{bin.bytes.value} bytes.

bin_bytes_value

Contains a positive integer representing the length, in bytes, of the binary aggregate value used by the various callback functions. Every byte sequence for this field must be exactly this length, and it also governs the length of the byte sequence specified by initial_value.

bin_merge_func

\[
\text{byte-sequence} = \text{bin_merge_func}(\text{byte-sequence}, \text{byte-sequence})
\]

Names a callback function which returns the result of merging two binary aggregate values into a new binary aggregate value. This merge function will often be addition; however, if the aggregate value is a bitmap, the result of merge function could be the union of the bitmaps. The function should take two byte sequence arguments and return a byte sequence, where all byte sequences are exactly \text{bin.bytes.value} bytes in length. If merging the aggregate values is not possible, the function should throw an exception. This function is used when the data structure used by \text{rwstats} or \text{rwuniq} runs out memory. When that happens, the application writes its current state to a temporary file, empties its buffers, and continues reading records. Once all records have been processed, the application needs to merge the temporary files to produce the final output. The bin_merge_func() function is used when merging these binary aggregate values.

bin_to_text_func

\[
\text{string} = \text{bin_to_text_func}(\text{byte-sequence})
\]

Names a callback function that takes a byte sequence representing an aggregate value as an argument and returns a printable representation of that aggregate value. The byte sequence input to bin_to_text_func() will be exactly \text{bin.bytes.value} bytes long. The length of the text should be no longer than the value specified by column_width. If the value returned from this function is not a string, the returned value is converted to a string by the Python str() function.

column_width_value

Contains a positive integer specifying the length of the longest textual field that the bin_to_text_func() callback function returns. This length will used as the column width when columnar output is requested.

bin_compare_func

\[
\text{int} = \text{bin_compare_func}(\text{byte-sequence}, \text{byte-sequence})
\]

Names a callback function that is called with two aggregate values, each represented as a byte sequence of exactly \text{bin.bytes.value} bytes. The function returns (1) an integer less than 0 if the first argument is less than the second, (2) an integer greater than 0 if the first is greater than the second, or (3) 0 if the two values are equal. This function is used by \text{rwstats} to sort the bins into top-N order.

description_string

Provides a string giving a brief description of the aggregate value, suitable for printing in --help-fields output. This argument is optional.

initial_value

Specifies a byte sequence representing the initial state of the binary aggregate value. This byte sequence must be of length \text{bin.bytes.value} bytes. If this argument is not specified, the aggregate value is set to a byte sequence containing \text{bin.bytes.value} null bytes.
initialize_func

initialize_func(). Names a callback function that is called after the command line arguments have been processed, and before opening the first file. This function is only called when --values includes agg.value_name. The function takes no arguments and its return value is ignored. This argument is optional.

Byte sequences

The rwgroup, rwsort, rwstats, and rwuniq programs make extensive use of "byte sequences" (a.k.a., "array of bytes", "byte strings", or "binary values") in their plug-in functions. The byte sequences are used in both key fields and aggregate values.

When used as key fields, the values can represent uniqueness or indicate sort order. Two records with the same byte sequence for a field will be considered identical with respect to that field. When sorting, the byte sequences are compared in network byte order. That is, the most significant byte is compared first, followed by the next-most-significant byte, etc. This equates to string comparison starting with the left-hand side of the string.

When used as an aggregate field, the byte sequences are expected to behave more like numbers, with the ability to take binary record and add a value to it, or to merge (e.g., add) two byte sequences outside the context of a SiLK record.

Every byte sequence has an associated length, which is passed into the register_field() function in the bin_bytes argument. The length determines how many values the byte sequence can represent. A byte sequence with a length of 1 can represent up to 256 unique values (from 0 to 255 inclusive). A byte sequence with a length of 2 can represent up to 65536 unique values (0 to 65535). To generalize, a byte sequence with a length of $n$ can represent up to $2^{8n}$ unique values (0 to $2^{8n}-1$).

How byte sequences are represented in Python depends on the version of Python. Python represents a sequence of characters using either the bytes type (introduced in 2.6) or the unicode type. The bytes type can encode byte sequences while the unicode type cannot. In Python 2, the str (string) type was an alias for bytes, so that any Python 2 string is in effect a byte sequence. In Python 3, str is an alias for unicode, thus Python 3 strings are unicode objects and cannot represent byte sequences.

Python does not make conversions between integers and byte sequences particularly natural. As a result, here are some pointers on how to do these conversions:

Use the bytes() and ord() methods

If you converting a single integer value that is less than 256, the easiest way to convert it to a byte sequence is to use the bytes() function; to convert it back, use the ord() function.

```python
seq = bytes([num])
num = ord(seq)
```

The bytes() function takes a list of integers between 0 and 255 inclusive, and returns a bytes sequence of the length of that list. To convert a single byte, use a list of a single element. The ord() function takes a byte sequence of a single byte and returns an integer between 0 and 255.

Note: In versions of Python earlier than 2.6, use the chr() function instead of the bytes() function. It takes a single number as its argument. chr() will work in Python 2.6 and 2.7 as well, but there are compatibility problems in Python 3.x.

Use the struct module
When the value you are converting to a byte sequence is 255 or greater, you have to go with another option. One of the simpler options is to use Python’s built-in `struct` module. With this module, you can encode a number or a set of numbers into a byte sequence and convert the result back using a `struct.Struct` object. Encoding the numbers to a byte sequence uses the object’s `pack()` method. To convert that byte sequence back to the number or set of numbers, use the object’s `unpack()` method. The length of the resulting byte sequences can be found in the `size` attribute of the `struct.Struct()` object. A formatting string is used to indicate how the numbers are encoded into binary. For example:

```python
import struct

# Set up the format for two 64-bit numbers
two64 = struct.Struct("!QQ")
# Encode two 64-bit numbers as a byte sequence
seq = two64.pack(num1, num2)
# Unpack a byte sequence back into two 64-bit numbers
(num1, num2) = two64.unpack(seq)
# Length of the encoded byte sequence
bin_bytes = two64.size
```

In the above, `Q` represents a single unsigned 64-bit number (an unsigned long long or quad). The `!` at the beginning of the string forces network byte order. (For sort comparison purposes, always pack in network byte order.)

Here is another example, which encodes a signed 16-bit integer and a floating point number:

```python
import struct

# Set up the format for a 16-bit signed integer and a float
obj = struct.Struct("!hf")
# Encode a 16-bit signed integer and a float as a byte sequence
seq = obj.pack(intval, floatval)
# Unpack a byte sequence back into a 16-bit signed integer and a float
(intval, floatval) = obj.unpack(seq)
# Length of the encoded byte sequence
bin_bytes = obj.size
```

Note that `unpack()` returns a sequence. When unpacking a single value, assign the result of `unpack` to `(variable_name,)`, as shown:

```python
import struct

u32 = struct.Struct("!I")
# Encode an unsigned 32-bit integer as a byte sequence
seq = u32.pack(num1)
# Unpack a byte sequence back into an unsigned 32-bit integer
(num1,) = struct.unpack(seq)
# Length of the encoded byte sequence
bin_bytes = u32.size
```
The full list of codes can be found in the Python library documentation for the `struct` module, [http://docs.python.org/library/struct.html](http://docs.python.org/library/struct.html).

**Note:** Python versions prior to 2.5 do not include support for the `struct.Struct` object. For older versions of Python, you have to use `struct`’s functional interface. For example:

```python
import struct

#Encode a 16-bit signed integer and a float as a byte sequence
seq = struct.pack("!hf", intval, floatval)

#Unpack a byte sequence back into a 16-bit signed integer and a float
(intval, floatval) = struct.unpack("!hf", seq)

#Length of the encoded byte sequence
bin_bytes = struct.calcsize("!hf")
```

This method works in Python 2.5 and above as well, but is inherently slower, as it requires re-evaluation of the format string for each packing and unpacking operation. Only use this if there is a need to inter-operate with older versions of Python.

**Use the array module**

The Python `array` module provides another way to create byte sequences. Beware that the `array` module does not provide an automatic way to encode the values in network byte order.

**OPTIONS**

The following options are available when the SiLK Python plug-in is used from `rwfilter`.

`--python-file=FILENAME`

Load the Python file `FILENAME`. The Python code may call `register_filter()` multiple times to define new partitioning functions that takes a `silk.RWRec` object as an argument. The return value of the function determines whether the record passes the filter. For backwards compatibility, if `register_filter()` is not called and a function named `rwfilter()` exists, that function is automatically registered as the filtering function. Multiple `--python-file` switches may be used to load multiple plug-ins.

`--python-expr=PYTHON_EXPRESSION`

Pass the SiLK Flow record if the result of the processing the record with the specified `PYTHON_EXPRESSION` is true. The expression is evaluated in the following context:

- The record is represented by the variable named `rec`, which is a `silk.RWRec` object.
- There is an implicit `from silk import *` in effect.

The following options are available when the SiLK Python plug-in is used from `rwcut`, `rwgroup`, `rwsort`, `rwstats`, or `rwuniq`:

`--python-file=FILENAME`

Load the Python file `FILENAME`. The Python code may call `register_field()` multiple times to define new fields for use by the application. When used with `rwstats` or `rwuniq`, the Python code may call `register_field()` multiple times to create new aggregate fields. Multiple `--python-file` switches may be used to load multiple plug-ins.
EXAMPLES

In the following examples, the dollar sign (\$) represents the shell prompt. The text after the dollar sign represents the command line. Lines have been wrapped for improved readability, and the back slash (\) is used to indicate a wrapped line.

**rwfilter --python-expr**

Suppose you want to find traffic destined to a particular host, 10.0.0.23, that is either ICMP or coming from 1434/udp. If you attempt to use:

```bash
$ rwfilter --daddr=10.0.0.23 --proto=1,17 --sport=1434 --pass=outfile.rw flowrec.rw
```

the **--sport** option will not match any of the ICMP traffic, and your result will not contain ICMP records. To avoid having to use two invocations of **rwfilter**, you can use the SiLK Python plugin to do the check in a single pass:

```bash
$ rwfilter --daddr=10.0.0.23 --proto=1,17 --python-expr 'rec.protocol==1 or rec.sport==1434' --pass=outfile.rw flowrec.rw
```

Since the Python code is slower than the C code used internally by **rwfilter**, we want to limit the number of records processed in Python as much as possible. We use the **rwfilter** switches to do the address check and protocol check, and in Python we only need to check whether the record is ICMP or if the source port is 1434 (if the record is not ICMP we know it is UDP because of the **--proto** switch).

**rwfilter --python-file**

To see all records whose protocol is different from the preceding record, use the following Python code. The code also prints a message to the standard output on completion.

```python
import sys

def filter(rec):
    global lastproto
    if rec.protocol != lastproto:
        lastproto = rec.protocol
        return True
    return False

def initialize():
    global lastproto
    lastproto = None

def finalize():
    sys.stdout.write("Finished processing records.\n")
```

58 SiLK -3.19.1
register_filter(filter, initialize = initialize, finalize = finalize)

The preceding file, if called lastproto.py, can be used like this:

$ rwfilter --python-file lastproto.py --pass=outfile.rw flowrec.rw

Note: Be careful when using a Python plug-in to write to the standard output, since the Python output could get intermingled with the output from --pass=stdout and corrupt the SiLK output file. In general, printing to the standard error is safer.

Command line switch

The following code registers the command line switch count-protocols. This switch is similar to the standard --protocol switch on rwfilter, in that it passes records whose protocol matches a value specified in a list. In addition, when rwfilter exits, the plug-in prints a count of the number of records that matched each specified protocol.

    import sys
    from silk.plugin import *

    pro_count = {}

    def proto_count(rec):
        global pro_count
        if rec.protocol in pro_count.keys():
            pro_count[rec.protocol] += 1
        return True
        return False

    def print_counts():
        for p, c in pro_count.iteritems():
            sys.stderr.write("%3d|\t%10d\n" % (p, c))

    def parse_protocols(protocols):
        global pro_count
        for p in protocols.split(","):  
            pro_count[int(p)] = 0
        register_filter(proto_count, finalize = print_counts)

    register_switch("count-protocols", handler=parse_protocols, 
                   help="Like --proto, but prints count of flow records")

When this code is saved to the file count-proto.py, it can be used with rwfilter as shown to get a count of TCP and UDP flow records:

$ rwfilter --start-date=2008/08/08 --type=out --python-file=count-proto.py --count-proto=6,17 --print-statistics=/dev/null

rwfilter does not know that the plug-in will be generating output, and rwfilter will complain unless an output switch is given, such as --pass or --print-statistics. Since our plug-in is printing the data we want, we send the output to /dev/null.
Create integer key field with simple API

This example creates a field that contains the sum of the source and destination port. While this value may not be interesting to display in rwcut, it provides a way to sort fields so traffic between two low ports will usually be sorted before traffic between a low port and a high port.

```python
def port_sum(rec):
    return rec.sport + rec.dport

register_int_field("port-sum", port_sum)
```

If the above code is saved in a file named `portsum.py`, it can be used to sort traffic prior to printing it (low-port to low-port will appear first):

```
$ rwfilter --start-date=2008/08/08 --type=out,outweb --proto=6,17 --pass=stdout
   | rwsort --python-file=portsum.py --fields=port-sum
   | rwcut
```

To see high-port to high-port traffic first, reverse the sort:

```
$ rwfilter --start-date=2008/08/08 --type=out,outweb --proto=6,17 --pass=stdout
   | rwsort --python-file=portsum.py --fields=port-sum --reverse
   | rwcut
```

Create IP key field with simple API

SiLK stores uni-directional flows. For network conversations that cross the network border, the source and destination hosts are swapped depending on the direction of the flow. For analysis, you often want to know the internal and external hosts.

The following Python plug-in file defines two new fields: `internal-ip` will display the destination IP for an incoming flow, and the source IP for an outgoing flow, and `external-ip` field shows the reverse.

```python
import silk

# for convenience, create lists of the types
in_types = ['in', 'inweb', 'innull', 'inicmp']
out_types = ['out', 'outweb', 'outnull', 'outicmp']

def internal(rec):
    """Returns the IP Address of the internal side of the connection""
    if rec.typename in out_types:
        return rec.sip
    else:
        return rec.dip
```

60 SiLK -3.19.1
def external(rec):
    "Returns the IP Address of the external side of the connection"
    if rec.typename in in_types:
        return rec.sip
    else:
        return rec.dip

register_ip_field("internal-ip", internal)
register_ip_field("external-ip", external)

If the above code is saved in a file named direction.py, it can be used to show the internal and external IP addresses and flow direction for all traffic on 1434/udp from Aug 8, 2008.

$ rwfilter --start-date=2008/08/08 --type=all
   --proto=17 --aport=1434 --pass=stdout
   | rwcut --python-file direction.py
   --fields internal-ip,external-ip,3-12

Create enumerated key field with simple API

This example expands the previous example. Suppose instead of printing the internal and external IP address, you wanted to group by the label associated with the internal and external addresses in a prefix map file. The pmapfilter(3) manual page specifies how to print labels for source and destination IP addresses, but it does not support internal and external IPs.

Here we take the previous example, add a command line switch to specify the path to a prefix map file, and have the internal and external functions return the label.

```
import silk

# for convenience, create lists of the types
in_types = ['in', 'inweb', 'innull', 'inicmp']
out_types = ['out', 'outweb', 'outnull', 'outicmp']

# handler for the --int-ext-pmap command line switch
def set_pmap(arg):
    global pmap
    pmap = silk.PrefixMap(arg)
    labels = pmap.values()
    width = max(len(x) for x in labels)
    register_enum_field("internal-label", internal, width, labels)
    register_enum_field("external-label", external, width, labels)

def internal(rec):
    "Returns the label for the internal side of the connection"
    global pmap
    if rec.typename in out_types:
        return pmap[rec.sip]
    else:
        return pmap[rec.dip]
```
def external(rec):
    """Returns the label for the external side of the connection""
    global pmap
    if rec.typename in in_types:
        return pmap[rec.sip]
    else:
        return pmap[rec.dip]

register_switch("int-ext-pmap", handler=set_pmap,
                help="Prefix map file for internal-label, external-label")

Assuming the above is saved in the file int-ext-pmap.py, the following will group the flows by the internal
and external labels contained in the file ip-map.pmap.

$ rwfilter --start-date=2008/08/08 --type=all
    --proto=17 --aport=1434 --pass=stdout
| rwuniq --python-file int-ext-pmap.py
    --int-ext-pmap ip-map.pmap
    --fields internal-label,external-label

Create minimum/maximum integer value field with simple API

The following example will create new aggregate fields to print the minimum and maximum byte values:

register_int_min_aggregator("min-bytes", lambda rec: rec.bytes,
    (1 << 32) - 1)
register_int_max_aggregator("max-bytes", lambda rec: rec.bytes,
    (1 << 32) - 1)

The lambda expression allows one to create an anonymous function. In this code, we need to return the
number of bytes for the given record, and we can easily do that with the anonymous function. Since the
SiLK bytes field is 32 bits, the maximum 32-bit number is passed the registration functions.

Assuming the code is stored in a file bytes.py, it can be used with rwuniq to see the minimum and maximum
byte counts for each source IP address:

$ rwuniq --python-file=bytes.py --fields=sip
    --values=records,bytes,min-bytes,max-bytes

Create IP key for rwcut with advanced API

This example is similar to the simple IP example above, but it uses the advanced API. It also creates another
field to indicate the direction of the flow, and it does not print the IPs when the traffic does not cross theorder. Note that this code has to determine the column width itself.

import silk, os
# for convenience, create lists of the types
in_types = ['in', 'inweb', 'innull', 'inicmp']
out_types = ['out', 'outweb', 'outnull', 'outicmp']
internal_only = ['int2int']
external_only = ['ext2ext']

# determine the width of the IP field depending on whether SiLK
# was compiled with IPv6 support, and allow the IP_WIDTH environment
# variable to override that width.
ip_len = 15
if silk.ipv6_enabled():
ip_len = 39
ip_len = int(os.getenv("IP_WIDTH", ip_len))

def cut_internal(rec):
    "Returns the IP Address of the internal side of the connection"
    if rec.typename in in_types:
        return rec.dip
    if rec.typename in out_types:
        return rec.sip
    if rec.typename in internal_only:
        return "both"
    if rec.typename in external_only:
        return "neither"
    return "unknown"

def cut_external(rec):
    "Returns the IP Address of the external side of the connection"
    if rec.typename in in_types:
        return rec.sip
    if rec.typename in out_types:
        return rec.dip
    if rec.typename in internal_only:
        return "neither"
    if rec.typename in external_only:
        return "both"
    return "unknown"

def internal_external_direction(rec):
    """Generates a string pointing from the sip to the dip, assuming
    internal is on the left, and external is on the right."""
    if rec.typename in in_types:
        return "<---"
    if rec.typename in out_types:
        return "--->"
    if rec.typename in internal_only:
        return "-><="
    if rec.typename in external_only:
        return "<-->"
    return "????"
register_field("internal-ip", column_width = ip_len,
    rec_to_text = cut_internal)
register_field("external-ip", column_width = ip_len,
    rec_to_text = cut_external)
register_field("int_to_ext", column_width = 4,
    rec_to_text = internal_external_direction)

The `cut_internal()` and `cut_external()` functions may return an IPAddr object instead of a string. For those cases, the Python `str()` function is invoked automatically to convert the IPAddr to a string.

If the above code is saved in a file named `direction.py`, it can be used to show the internal and external IP addresses and flow direction for all traffic on 1434/udp from Aug 8, 2008.

```
$ rwfilter --start-date=2008/08/08 --type=all --proto=17 --aport=1434 --pass=stdout | rwcut --python-file direction.py --fields internal-ip,int_to_ext,external-ip,3-12
```

Create integer key field for rwsort with the advanced API

The following example Python plug-in creates one new field, `lowest_port`, for use in rwsort. Using this field will sort records based on the lesser of the source port or destination port; for example, flows where either the source or destination port is 22 will occur before flows where either port is 25. This example shows using the Python struct module with multiple record attributes.

```
import struct

portpair = struct.Struct("!HH")

def lowest_port(rec):
    if rec.sport < rec.dport:
        return portpair.pack(rec.sport, rec.dport)
    else:
        return portpair.pack(rec.dport, rec.sport)

register_field("lowest_port", bin_bytes = portpair.size,
    rec_to_bin = lowest_port)
```

To use this example to sort the records in `flowrec.rw`, one saves the code to the file `sort.py` and uses it as shown:

```
$ rwsort --python-file=sort.py --fields=lowest_port flowrec.rw > outfile.rw
```

Create integer key for rwstats and rwuniq with advanced API

The following example defines two key fields for use by rwstats or rwuniq: `prefixed-sip` and `prefixed-dip`. Using these fields, the user can count flow records based on the source and/or destination IPv4 address blocks (CIDR blocks). The default CIDR prefix is 16, but it can be changed by specifying the --prefix switch that the example creates. This example uses the Python struct module to convert between the IP address and a binary string.

```
import struct

portpair = struct.Struct("!HH")

def lowest_port(rec):
    if rec.sport < rec.dport:
        return portpair.pack(rec.sport, rec.dport)
    else:
        return portpair.pack(rec.dport, rec.sport)

register_field("lowest_port", bin_bytes = portpair.size,
    rec_to_bin = lowest_port)
```
import os, struct
from silk import *

def set_mask(prefix):
    global mask
    mask = 0xFFFFFFFF
    # the value we are handed is a string
    prefix = int(prefix)
    if 0 < prefix < 32:
        mask = mask ^ (mask >> prefix)

    # Convert from an IPv4Addr to a byte sequence
    def cidr_to_bin(ip):
        if ip.is_ipv6():
            raise ValueError, "Does not support IPv6"
        return u32.pack(int(ip) & mask)

    # Convert from a byte sequence to an IPv4Addr
    def cidr_bin_to_text(string):
        (num,) = u32.unpack(string)
        return IPv4Addr(num)

    register_field("prefixed-sip", column_width = 15,
                   rec_to_bin = lambda rec: cidr_to_bin(rec.sip),
                   bin_to_text = cidr_bin_to_text,
                   bin_bytes = u32.size)

    register_field("prefixed-dip", column_width = 15,
                   rec_to_bin = lambda rec: cidr_to_bin(rec.dip),
                   bin_to_text = cidr_bin_to_text,
                   bin_bytes = u32.size)

    register_switch("prefix", handler=set_mask,
                   help="Set prefix for prefixed-sip/prefixed-dip fields")

    set_mask(default_prefix)

    The lambda expression allows one to create an anonymous function. In this code, the lambda function
    is used to pass the appropriate IP address into the cidr_to_bin() function. To write the code without the
    lambda would require separate functions for the source and destination IP addresses:

    def sip_cidr_to_bin(rec):
        return cidr_to_bin(rec.sip)

    def dip_cidr_to_bin(rec):
        return cidr_to_bin(rec.dip)
The **lambda** expression helps to simplify the code.

If the code is saved in the file `mask.py`, it can be used as follows to count the number of flow records seen in the /8 of each source IP address. The flow records are read from `flowrec.rw`. The `--ipv6-policy=ignore` switch is used to restrict processing to IPv4 addresses.

```bash
$ rwuniq --ipv6-policy=ignore --python-file mask.py \
   --prefix 8 --fields prefixed-sip flowrec.rw
```

**Create new average bytes value field for rwstats and rwuniq**

The following example creates a new aggregate value that can be used by **rwstats** and **rwuniq**. The value is `avg-bytes`, a value that calculates the average number of bytes seen across all flows that match the key. It does this by maintaining running totals of the byte count and number of flows.

```python
import struct

fmt = struct.Struct("QQ")
initial = fmt.pack(0, 0)
textsize = 15
textformat = "%%%d.2f" % textsize

# add byte and flow count from 'rec' to 'current'
def avg_bytes(rec, current):
    (total, count) = fmt.unpack(current)
    return fmt.pack(total + rec.bytes, count + 1)

# return printable representation
def avg_to_text(bin):
    (total, count) = fmt.unpack(bin)
    return textformat % (float(total) / count)

# merge two encoded values.
def avg_merge(rec1, rec2):
    (total1, count1) = fmt.unpack(rec1)
    (total2, count2) = fmt.unpack(rec2)
    return fmt.pack(total1 + total2, count1 + count2)

# compare two encoded values
def avg_compare(rec1, rec2):
    (total1, count1) = fmt.unpack(rec1)
    (total2, count2) = fmt.unpack(rec2)
    # Python 2:
    #return cmp((float(total1) / count1), (float(total2) / count2))
    # Python 3:
    avg1 = float(total1) / count1
    avg2 = float(total2) / count2
    if avg1 < avg2:
        return -1
    return avg1 > avg2
```
register_field("avg-bytes",
    column_width = textsize,
    bin_bytes = fmt.size,
    add_rec_to_bin = avg_bytes,
    bin_to_text = avg_to_text,
    bin_merge = avg_merge,
    bin_compare = avg_compare,
    initial_value = initial)

To use this code, save it as *avg-bytes.py*, specify the name of the Python file in the `--python-file` switch,
and list the field in the `--values` switch:

```
$ rwuniq --python-file=avg-bytes.py --fields=sip --values=avg-bytes infile.rw
```

This particular example will compute the average number of bytes per flow for each distinct source IP address
in the file *infile.rw*.

**Create integer key field for all tools that use fields**

The following example Python plug-in file defines two fields, *sport-service* and *dport-service*. These
fields convert the source port and destination port to the name of the "service" as defined in the file
*/etc/services*; for example, port 80 is converted to "http". This plug-in can be used by any of *rwcut*,
*rwgroup*, *rwsort*, *rwstats*, or *rwuniq*.

```
import os,socket,struct

u16 = struct.Struct("!H")

# utility function to convert number to a service name,
# or to a string if no service is defined
def num_to_service(num):
    try:
        serv = socket.getservbyport(num)
    except socket.error:
        serv = "%d" % num
    return serv

# convert the encoded port to a service name
def bin_to_service(bin):
    (port,) = u16.unpack(bin)
    return num_to_service(port)

# width of service columns can be specified with the
# SERVICE_WIDTH environment variable; default is 12
col_width = int(os.getenv("SERVICE_WIDTH", 12))

register_field("sport-service", bin_bytes = u16.size,
    column_width = col_width,
    initial_value = initial)
```
rec_to_text = lambda rec: num_to_service(rec.sport),
rec_to_bin = lambda rec: u16.pack(rec.sport),
bin_to_text = bin_to_service)

register_field("dport-service", bin_bytes = u16.size,
column_width = col_width,
rec_to_text = lambda rec: num_to_service(rec.dport),
rec_to_bin = lambda rec: u16.pack(rec.dport),
bin_to_text = bin_to_service)

If this file is named service.py, it can be used by rwcut to print the source port and its service:

$ rwcut --python-file service.py \
   --fields sport,sport-service flowrec.rw

Although the plug-in can be used with rwsort, the records will be sorted in the same order as the numerical source port or destination port.

$ rwsort --python-file service.py \
   --fields sport-service flowrec.rw > outfile.rw

When used with rwuniq, it can count flows, bytes, and packets indexed by the service of the destination port:

$ rwuniq --python-file service.py --fields dport-service \
   --values=flows,bytes,packets flowrec.rw

Create human-readable fields for all tools that use fields

The following example adds two fields, hu-bytes and hu-packets, which can be used as either key fields or aggregate value fields. The example uses the formatting capabilities of netsa-python (http://tools.netsa.cert.org/netsa-python/index.html) to present the bytes and packets fields in a more human-friendly manner.

When used as a key, the hu-bytes field presents the value 1234567 as 1205.6Ki or as 1234.6k when the HUMAN_USE_BINARY environment variable is set to False.

When used as a key, the hu-packets field adds a comma (or the character specified by the HUMAN_THOUSANDS_SEP environment variable) to the display of the packets field. The value 1234567 becomes 1,234,567.

The hu-bytes and hu-packets fields can also be used as aggregate value fields, in which case they compute the sum of the bytes and packets, respectively, and display it as for the key field.

The code for the plug-in is shown here, and an example of using the plug-in follows the code.

```python
import silk, silk.plugin
import os, struct
from netsa.data.format import num_prefix, num_fixed
```
# Whether the use Base-2 (True) or Base-10 (False) values for Kibi/Mebi/Gibi/Tebi/... vs Kilo/Mega/Giga/Tera/...
use_binary = True
if (os.getenv("HUMAN_USE_BINARY")):
    if (os.getenv("HUMAN_USE_BINARY").lower() == "false"
        or os.getenv("HUMAN_USE_BINARY") == "0"):
        use_binary = False
    else:
        use_binary = True

# Character to use for Thousands separator
thousands_sep = ','
if (os.getenv("HUMAN_THOUSANDS_SEP")):
    thousands_sep = os.getenv("HUMAN_THOUSANDS_SEP")

# Number of significant digits
sig_fig=5

# Use a 64-bit number for packing the bytes or packets data
fmt = struct.Struct("Q")
initial = fmt.pack(0)

### Bytes functions
# add_rec_to_bin
def hu_ar2b_bytes(rec, current):
    global fmt
    (cur,) = fmt.unpack(current)
    return fmt.pack(cur + rec.bytes)

# rec_to_binary
def hu_r2b_bytes(rec):
    global fmt
    return fmt.pack(rec.bytes)

# bin_to_text
def hu_b2t_bytes(current):
    global use_binary, sig_fig, fmt
    (cur,) = fmt.unpack(current)
    return num_prefix(cur, use_binary=use_binary, sig_fig=sig_fig)

# rec_to_text
def hu_r2t_bytes(rec):
    global use_binary, sig_fig
    return num_prefix(rec.bytes, use_binary=use_binary, sig_fig=sig_fig)

### Packets functions
# add_rec_to_bin
def hu_ar2b_packets(rec, current):
    global fmt
    (cur,) = fmt.unpack(current)
    return fmt.pack(cur + rec.packets)

# rec_to_binary
def hu_r2b_packets(rec):
    global fmt
    return fmt.pack(rec.packets)

# bin_to_text
def hu_b2t_packets(current):
    global use_binary, sig_fig, fmt
    (cur,) = fmt.unpack(current)
    return num_prefix(cur, use_binary=use_binary, sig_fig=sig_fig)

# rec_to_text
def hu_r2t_packets(rec):
    global use_binary, sig_fig
    return num_prefix(rec.bytes, use_binary=use_binary, sig_fig=sig_fig)
# rec_to_binary
def hu_r2b_packets(rec):
    global fmt
    return fmt.pack(rec.packets)

# bin_to_text
def hu_b2t_packets(current):
    global thousands_sep, fmt
    (cur,) = fmt.unpack(current)
    return num_fixed(cur, dec_fig=0, thousands_sep=thousands_sep)

# rec_to_text
def hu_r2t_packets(rec):
    global thousands_sep
    return num_fixed(rec.packets, dec_fig=0, thousands_sep=thousands_sep)

### Non-specific functions

# bin_compare
def hu_bin_compare(cur1, cur2):
    if (cur1 < cur2):
        return -1
    return (cur1 > cur2)

# bin_merge
def hu_bin_merge(current1, current2):
    global fmt
    (cur1,) = fmt.unpack(current1)
    (cur2,) = fmt.unpack(current2)
    return fmt.pack(cur1 + cur2)

### Register the fields
register_field("hu-bytes", column_width=10, bin_bytes=fmt.size,
                rec_to_text=hu_r2t_bytes, rec_to_bin=hu_r2b_bytes,
                bin_to_text=hu_b2t_bytes, add_rec_to_bin=hu_ar2b_bytes,
                bin_merge=hu_bin_merge, bin_compare=hu_bin_compare,
                initial_value=initial)

register_field("hu-packets", column_width=10, bin_bytes=fmt.size,
                rec_to_text=hu_r2t_packets, rec_to_bin=hu_r2b_packets,
                bin_to_text=hu_b2t_packets, add_rec_to_bin=hu_ar2b_packets,
                bin_merge=hu_bin_merge, bin_compare=hu_bin_compare,
                initial_value=initial)

This shows an example of the plug-in’s invocation and output when the code below is stored in the file human.py.

$ rwstats --count=5 --no-percent --python-file=human.py \
--fields=proto,hu-bytes,hu-packets \ 
--values=records,hu-bytes,hu-packets data.rw
INPUT: 501876 Records for 305417 Bins and 501876 Total Records
Identifying SMTP Servers

To demonstrate the use of \texttt{--python-file} in \texttt{rwfilter(1)}, we walk through a Python plug-in script that evaluates the behavior of a set of IP addresses and determines if the host is likely to be an SMTP server or relay. We expect (based on traffic studies) that more than 85\% of a legitimate SMTP server’s activity is devoted to sending or providing mail. If we find that the host exhibits this behavior, we include the IP address in a set called \texttt{SMTP.set}. Regardless of if the IP address is included in the set, we pass all records that appear to be legitimate mail flows.

We run the \texttt{rwfilter} command as follows:

\begin{verbatim}
$ rwfilter --start-date=2008/4/21 --end-date=2008/4/21 \\
    --type=out,outweb --sipset=possible_SMTP_servers.set \\
    --python-file=SMTP.py --print-statistics
\end{verbatim}

This command first collects all records of type \texttt{out} and \texttt{outweb} that have a start date on April 21, 2008. Since there are no additional command line options to filter records, all records are passed to the \texttt{rwfilter(rec)} function in \texttt{SMTP.py}. \texttt{rec} is an instance of the object \texttt{RWRec}, which represents the record being passed.

The function \texttt{rwfilter(rec)} in \texttt{SMTP.py} begins by importing the global variable \texttt{counts} and \texttt{smtpports}. \texttt{counts} is a dictionary indexed by source IP address and contains an array of size two, where the first element is the total number of bytes that the IP address has transferred and the second element is the number of bytes that the source address has transferred that are likely to be related to mail delivery.

Using the source IP address from the record, the function retrieves the current byte counts from the \texttt{counts} dictionary. If this is the first occurrence of the IP address, a new entry is added. The function then adds the byte count of this record to the total byte count and determines if the record is a mail delivery message. If it is a mail message, the function adds the bytes to the total of bytes transferred as mail and returns \texttt{True}. Otherwise, a value of \texttt{False} is returned.

After \texttt{rwfilter} processes all records it calls the \texttt{finalize()} function, which evaluates the collection of IP addresses. If the percentage of bytes that the host transferred in mail operations is greater than 85\% of the total bytes transferred, the IP address is added to a final set of SMTP servers. The final set of SMTP servers is then saved to the \texttt{SMTP.set} file, and \texttt{rwfilter} exits.

```python
from silk import *

# Collection of ports commonly used by SMTP servers
smtpports = set([25, 109, 110, 143, 220, 273, 993, 995, 113])

# Minimum percentage of mail traffic before being considered a mail server
threshold = 0.85

# Collection of byte counts
counts = dict()
```
# This function is run over all records.
# Input: An instance of the RWRec class representing the
# current record being processed
# Output: True or false value indicating if the record passes
# or fails the filter
def rwfilter(rec):
    # Import the global variables needed for processing the record
    global smtpports, counts

    # Pull data from the record
    sip = rec.sip
    bytes = rec.bytes

    # Get a reference to the current data on the IP address in question
    data = counts.setdefault(sip, [0, 0])

    # Update the total byte count for the IP address
    data[0] += bytes

    # Is the flow mail related? If so add the byte count to the mail bytes
    if (rec.protocol == 6 and rec.sport in smtpports and
        rec.packets > 3 and rec.bytes > 120):
        data[1] += bytes
        return True

    # If not mail related, fail the record
    return False

# This is run after all records have been processed
def finalize():
    # Import the global variables needed to evaluate the results
    global counts, threshold

    # The IP set of SMTP servers
    smtp = IPSet()

    # Iterate through all of the IP addresses.
    for ip, data in counts.iteritems():
        if (float(data[1]) / data[0]) > threshold:
            smtp.add(ip)

    # Generate the IPset of all smtp servers.
    smtp.save('smtp.set')

    # Register these functions with rwfilter
    register_filter(rwfilter, finalize=finalize)
UPGRADING LEGACY PLUGINS

Some functions were marked as deprecated in SiLK 2.0, and have been removed in SiLK 3.0.

Prior to SiLK 2.0, the `register_field()` function was called `register_plugin_field()`, and it had the following signature:

```
register_plugin_field(field_name, [bin_len=bin_bytes_value], [bin_to_text=bin_to_text_func],
                      [text_len=column_width_value], [rec_to_bin=rec_to_bin_func], [rec_to_text=rec_to_text_func])
```

To convert from `register_plugin_field` to `register_field`, change `text_len` to `column_width`, and change `bin_len` to `bin_bytes`. (Even older code may use `field_len`; this should be changed to `column_width` as well.)

The `register_filter()` function was introduced in SiLK 2.0. In versions of SiLK prior to SiLK 3.0, when `rwfilter` was invoked with `--python-file` and the named Python file did not call `register_filter()`, `rwfilter` would search the Python input for functions named `rwfilter()` and `finalize()`. If it found the `rwfilter()` function, `rwfilter` would act as if the file contained:

```
register_filter(rwfilter, finalize=finalize)
```

To update your pre-SiLK 2.0 `rwfilter` plug-ins, simply add the above line to your Python file.

ENVIRONMENT

PYTHONPATH

This environment variable is used by Python to locate modules. When `--python-file` or `--python_expr` is specified, the application must load the Python files that comprise the PySiLK package, such as `silk/__init__.py`. If this `silk/` directory is located outside Python's normal search path (for example, in the SiLK installation tree), it may be necessary to set or modify the PYTHONPATH environment variable to include the parent directory of `silk/` so that Python can find the PySiLK module.

PYTHONVERBOSE

If the SiLK Python extension or plug-in fails to load, setting this environment variable to a non-empty string may help you debug the issue.

SILK_PYTHON_TRACEBACK

When set, Python plug-ins will output trace back information regarding Python errors to the standard error.

SEE ALSO

`pysilk(3)`, `rwfilter(1)`, `rwcut(1)`, `rwgroup(1)`, `rwsort(1)`, `rwstats(1)`, `rwuniq(1)`, `pmapfilter(3)`, `silk(7)`, `python(1)`, `http://docs.python.org/`
Appendix A

License

Use of the SiLK system and related source code is subject to the terms of the following licenses:

- GNU General Public License (GPL) Rights pursuant to Version 2, June 1991
- Government Purpose License Rights (GPLR) pursuant to DFARS 252.227.7013

NO WARRANTY

ANY INFORMATION, MATERIALS, SERVICES, INTELLECTUAL PROPERTY OR OTHER PROPERTY OR RIGHTS GRANTED OR PROVIDED BY CARNEGIE MELLON UNIVERSITY PURSUANT TO THIS LICENSE (HEREINAFTER THE "DELIVERABLES") ARE ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, INFORMATIONAL CONTENT, NONINFRINGEMENT, OR ERROR-FREE OPERATION. CARNEGIE MELLON UNIVERSITY SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS LOSS OF PROFITS OR INABILITY TO USE SAID INTELLECTUAL PROPERTY, UNDER THIS LICENSE, REGARDLESS OF WHETHER SUCH PARTY WAS AWARE OF THE POSSIBILITY OF SUCH DAMAGES. LICENSEE AGREES THAT IT WILL NOT MAKE ANY WARRANTY ON BEHALF OF CARNEGIE MELLON UNIVERSITY, EXPRESS OR IMPLIED, TO ANY PERSON CONCERNING THE APPLICATION OF OR THE RESULTS TO BE OBTAINED WITH THE DELIVERABLES UNDER THIS LICENSE.

Licensee hereby agrees to defend, indemnify, and hold harmless Carnegie Mellon University, its trustees, officers, employees, and agents from all claims or demands made against them (and any related losses, expenses, or attorney’s fees) arising out of, or relating to Licensee’s and/or its sub licensees’ negligent use or willful misuse of or negligent conduct or willful misconduct regarding the Software, facilities, or other rights or assistance granted by Carnegie Mellon University under this License, including, but not limited to, any claims of product liability, personal injury, death, damage to property, or violation of any laws or regulations.

Carnegie Mellon University Software Engineering Institute authored documents are sponsored by the U.S. Department of Defense under Contract FA8702-15-D-0002. Carnegie Mellon University retains copyrights in all material produced under this contract. The U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce these documents, or allow others to do so, for U.S. Government purposes only pursuant to the copyright license under the contract clause at 252.227.7013.