NAME

SIFTR - Statistical Information For TCP Research

SYNOPSIS

To load the driver as a module at run-time, run the following command as root:

```
kldload siftr
```

Alternatively, to load the driver as a module at boot time, add the following line into the loader.conf(5) file:

```
siftr_load="YES"
```

DESCRIPTION

The **SIFTR** (*S*tatistical *I*nformation *F*or *T*CP *R*esearch) kernel module logs a range of statistics on active TCP connections to a log file. It provides the ability to make highly granular measurements of TCP connection state, aimed at system administrators, developers and researchers.

Compile-time Configuration

The default operation of **SIFTR** is to capture IPv4 TCP/IP packets. **SIFTR** can be configured to support IPv4 and IPv6 by uncommenting:

```
CFLAGS+=-DSIFTR_IPV6
```

in <sys/modules/siftr/Makefile> and recompiling.

In the IPv4-only (default) mode, standard dotted decimal notation (e.g. "136.186.229.95") is used to format IPv4 addresses for logging. In IPv6 mode, standard dotted decimal notation is used to format IPv4 addresses, and standard colon-separated hex notation (see RFC 4291) is used to format IPv6 addresses (e.g. "fd00::2") for logging.

Run-time Configuration

SIFTR utilises the sysctl(8) interface to export its configuration variables to user-space. The following variables are available:

net.inet.siftr.enabled

controls whether the module performs its measurements or not. By default, the value is set to 0, which means the module will not be taking any measurements. Having the module loaded with *net.inet.siftr.enabled* set to 0 will have no impact on the performance of the network stack, as the packet filtering hooks are only

inserted when *net.inet.siftr.enabled* is set to 1.

net.inet.siftr.ppl
controls how many inbound/outbound packets for a given TCP connection will cause a log message to be generated for the connection. By default, the value is set to 1, which means the module will log a message for every packet of every TCP connection. The value can be set to any integer in the range [1,2^32], and can be changed at any time, even while the module is enabled.

net.inet.siftr.logfile
controls the path to the file that the module writes its log messages to. By default, the file /var/log/siftr.log is used. The path can be changed at any time, even while the module is enabled.

net.inet.siftr.port_filter
controls on which source or destination port **SIFTR** should capture. By default, the value is set to 0, which means all ports are eligible for logging. Set to any other value, only packets where either the source or destination port is equal to this number are logged.

Log Format

A typical **SIFTR** log file will contain 3 different types of log message. All messages are written in plain ASCII text.

Note: The "\" present in the example log messages in this section indicates a line continuation and is not part of the actual log message.

The first type of log message is written to the file when the module is enabled and starts collecting data from the running kernel. The text below shows an example module enable log. The fields are tab delimited key-value pairs which describe some basic information about the system.

```
enable_time_secs=1685191807    enable_time_usecs=160752 \
siftrver=1.3.0    sysname=FreeBSD    sysver=1400089    ipmode=4
```

Field descriptions are as follows:

enable_time_secs
time at which the module was enabled, in seconds since the UNIX epoch.

enable_time_usecs

time at which the module was enabled, in microseconds since enable_time_secs.

siftrver version of **SIFTR**.

sysname operating system name.

sysver operating system version.

ipmode IP mode as defined at compile time. An ipmode of "4" means IPv6 is not supported and IP addresses are logged in regular dotted quad format. An ipmode of "6" means IPv6 is supported, and IP addresses are logged in dotted quad or hex format, as described in the "Compile-time Configuration" subsection.

The second type of log message is written to the file when a data log message is generated. The text below shows an example data log triggered by an IPv4 TCP/IP packet. The data is CSV formatted.

```
o,1685191814.185109,10.1.1.2,32291,10.1.1.3,5001,1073725440, \
14480,2,65160,65700,7,9,4,1460,1000,1,16778209,230000,33580,0, \
65700,0,0,0,86707916,130
```

Field descriptions are as follows:

1 Direction of packet that triggered the log message. Either "i" for in, or "o" for out.

2 Time at which the packet that triggered the log message was processed by the pfil(9) hook function, in seconds and microseconds since the UNIX epoch.

3 The IPv4 or IPv6 address of the local host, in dotted quad (IPv4 packet) or colon-separated hex (IPv6 packet) notation.

4 The TCP port that the local host is communicating via.

5 The IPv4 or IPv6 address of the foreign host, in dotted quad (IPv4 packet) or colon-separated hex (IPv6 packet) notation.

6 The TCP port that the foreign host is communicating via.

7 The slow start threshold for the flow, in bytes.

8 The current congestion window for the flow, in bytes.

9 The current state of the t_flags2 field for the flow.

10 The current sending window for the flow, in bytes. The post scaled value is reported.

11 The current receive window for the flow, in bytes. The post scaled value is always reported.

12 The current window scaling factor for the sending window.

13 The current window scaling factor for the receiving window.

14 The current state of the TCP finite state machine, as defined in <*netinet/tcp_fsm.h*>.

15 The maximum segment size for the flow, in bytes.

16 The current smoothed RTT estimate for the flow, in units of microsecond.

17 SACK enabled indicator. 1 if SACK enabled, 0 otherwise.

18 The current state of the TCP flags for the flow. See <*netinet/tcp_var.h*> for information about the various flags.

19 The current retransmission timeout length for the flow, in units microsecond.

20 The current size of the socket send buffer in bytes.

21 The current number of bytes in the socket send buffer.

22 The current size of the socket receive buffer in bytes.

23 The current number of bytes in the socket receive buffer.

24 The current number of unacknowledged bytes in-flight. Bytes acknowledged via SACK are not excluded from this count.

25 The current number of segments in the reassembly queue.

26 Flowid for the connection. A caveat: Zero ’0’ either represents a valid flowid or a

default value when it’s not being set. There is no easy way to differentiate without looking at actual network interface card and drivers being used.

27 Flow type for the connection. Flowtype defines which protocol fields are hashed to produce the flowid. A complete listing is available in *sys/mbuf.h* under M_HASHTYPE_*.

The third type of log message is written to the file when the module is disabled and ceases collecting data from the running kernel. The text below shows an example module disable log. The fields are tab delimited key-value pairs which provide statistics about operations since the module was most recently enabled.

```
disable_time_secs=1685191816    disable_time_usecs=629397 \
num_inbound_tcp_pkts=10    num_outbound_tcp_pkts=10 \
total_tcp_pkts=20    num_inbound_skipped_pkts_malloc=0 \
num_outbound_skipped_pkts_malloc=0    num_inbound_skipped_pkts_tcpcb=2 \
num_outbound_skipped_pkts_tcpcb=2    num_inbound_skipped_pkts_inpcb=0 \
num_outbound_skipped_pkts_inpcb=0    total_skipped_tcp_pkts=4 \
flow_list=10.1.1.2;32291-10.1.1.3;5001,10.1.1.2;58544-10.1.1.3;5001,
```

Field descriptions are as follows:

disable_time_secs
Time at which the module was disabled, in seconds since the UNIX epoch.

disable_time_usecs
Time at which the module was disabled, in microseconds since disable_time_secs.

num_inbound_tcp_pkts
Number of TCP packets that traversed up the network stack. This only includes inbound TCP packets during the periods when **SIFTR** was enabled.

num_outbound_tcp_pkts
Number of TCP packets that traversed down the network stack. This only includes outbound TCP packets during the periods when **SIFTR** was enabled.

total_tcp_pkts The summation of num_inbound_tcp_pkts and num_outbound_tcp_pkts.

num_inbound_skipped_pkts_malloc
Number of inbound packets that were not processed because of failed **malloc**()

calls.

num_outbound_skipped_pkts_malloc
Number of outbound packets that were not processed because of failed **malloc**() calls.

num_inbound_skipped_pkts_tcpcb
Number of inbound packets that were not processed because of failure to find the TCP control block associated with the packet.

num_outbound_skipped_pkts_tcpcb
Number of outbound packets that were not processed because of failure to find the TCP control block associated with the packet.

num_inbound_skipped_pkts_inpcb
Number of inbound packets that were not processed because of failure to find the IP control block associated with the packet.

num_outbound_skipped_pkts_inpcb
Number of outbound packets that were not processed because of failure to find the IP control block associated with the packet.

total_skipped_tcp_pkts
The summation of all skipped packet counters.

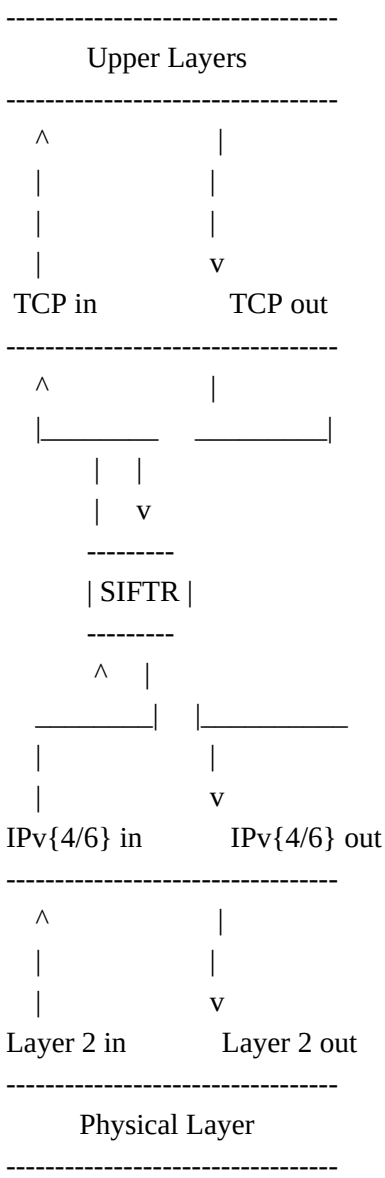
flow_list A CSV list of TCP flows that triggered data log messages to be generated since the module was loaded. Each flow entry in the CSV list is formatted as "local_ip;local_port-foreign_ip;foreign_port". If there are no entries in the list (i.e., no data log messages were generated), the value will be blank. If there is at least one entry in the list, a trailing comma will always be present.

The total number of data log messages found in the log file for a module enable/disable cycle should equate to total_tcp_pkts - total_skipped_tcp_pkts.

IMPLEMENTATION NOTES

SIFTR hooks into the network stack using the pfil(9) interface. In its current incarnation, it hooks into the AF_INET/AF_INET6 (IPv4/IPv6) pfil(9) filtering points, which means it sees packets at the IP layer of the network stack. This means that TCP packets inbound to the stack are intercepted before they have been processed by the TCP layer. Packets outbound from the stack are intercepted after they have been processed by the TCP layer.

The diagram below illustrates how **SIFTR** inserts itself into the stack.



SIFTR uses the alq(9) interface to manage writing data to disk.

At first glance, you might mistakenly think that **SIFTR** extracts information from individual TCP packets. This is not the case. **SIFTR** uses TCP packet events (inbound and outbound) for each TCP flow originating from the system to trigger a dump of the state of the TCP control block for that flow. With the PPL set to 1, we are in effect sampling each TCP flow's control block state as frequently as flow packets enter/leave the system. For example, setting PPL to 2 halves the sampling rate i.e., every second flow packet (inbound OR outbound) causes a dump of the control block state.

The distinction between interrogating individual packets versus interrogating the control block is important, because **SIFTR** does not remove the need for packet capturing tools like tcpdump(1). **SIFTR** allows you to correlate and observe the cause-and-affect relationship between what you see on the wire (captured using a tool like tcpdump(1)) and changes in the TCP control block corresponding to the flow of interest. It is therefore useful to use **SIFTR** and a tool like tcpdump(1) to gather the necessary data to piece together the complete picture. Use of either tool on its own will not be able to provide all of the necessary data.

As a result of needing to interrogate the TCP control block, certain packets during the lifecycle of a connection are unable to trigger a **SIFTR** log message. The initial handshake takes place without the existence of a control block or the complete initialization of the control block, and the final ACK is exchanged when the connection is in the TIMEWAIT state.

SIFTR was designed to minimise the delay introduced to packets traversing the network stack. This design called for a highly optimised and minimal hook function that extracted the minimal details necessary whilst holding the packet up, and passing these details to another thread for actual processing and logging.

This multithreaded design does introduce some contention issues when accessing the data structure shared between the threads of operation. When the hook function tries to place details in the structure, it must first acquire an exclusive lock. Likewise, when the processing thread tries to read details from the structure, it must also acquire an exclusive lock to do so. If one thread holds the lock, the other must wait before it can obtain it. This does introduce some additional bounded delay into the kernel’s packet processing code path.

In some cases (e.g., low memory, connection termination), TCP packets that enter the **SIFTR** pfil(9) hook function will not trigger a log message to be generated. **SIFTR** refers to this outcome as a "skipped packet". Note that **SIFTR** always ensures that packets are allowed to continue through the stack, even if they could not successfully trigger a data log message. **SIFTR** will therefore not introduce any packet loss for TCP/IP packets traversing the network stack.

Important Behaviours

The behaviour of a log file path change whilst the module is enabled is as follows:

1. Attempt to open the new file path for writing. If this fails, the path change will fail and the existing path will continue to be used.

2. Assuming the new path is valid and opened successfully:

   - Flush all pending log messages to the old file path.

- Close the old file path.
- Switch the active log file pointer to point at the new file path.
- Commence logging to the new file.

During the time between the flush of pending log messages to the old file and commencing logging to the new file, new log messages will still be generated and buffered. As soon as the new file path is ready for writing, the accumulated log messages will be written out to the file.

EXAMPLES

To enable the module’s operations, run the following command as root: sysctl net.inet.siftr.enabled=1

To change the granularity of log messages such that 1 log message is generated for every 10 TCP packets per connection, run the following command as root: sysctl net.inet.siftr.ppl=10

To change the log file location to /tmp/siftr.log, run the following command as root: sysctl net.inet.siftr.logfile=/tmp/siftr.log

SEE ALSO

tcpdump(1), tcp(4), sysctl(8), alq(9), pfil(9)

ACKNOWLEDGEMENTS

Development of this software was made possible in part by grants from the Cisco University Research Program Fund at Community Foundation Silicon Valley, and the FreeBSD Foundation.

HISTORY

SIFTR first appeared in FreeBSD 7.4 and FreeBSD 8.2.

SIFTR was first released in 2007 by Lawrence Stewart and James Healy whilst working on the NewTCP research project at Swinburne University of Technology’s Centre for Advanced Internet Architectures, Melbourne, Australia, which was made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley. More details are available at:

http://caia.swin.edu.au/urp/newtcp/

Work on **SIFTR** v1.2.x was sponsored by the FreeBSD Foundation as part of the "Enhancing the FreeBSD TCP Implementation" project 2008-2009. More details are available at:

https://www.freebsdfoundation.org/

http://caia.swin.edu.au/freebsd/etcp09/

AUTHORS

SIFTR was written by Lawrence Stewart <*lstewart@FreeBSD.org*> and James Healy <*jimmy@deefa.com*>.

This manual page was written by Lawrence Stewart <*lstewart@FreeBSD.org*>.

BUGS

Current known limitations and any relevant workarounds are outlined below:

- The internal queue used to pass information between the threads of operation is currently unbounded. This allows **SIFTR** to cope with bursty network traffic, but sustained high packet-per-second traffic can cause exhaustion of kernel memory if the processing thread cannot keep up with the packet rate.

- If using **SIFTR** on a machine that is also running other modules utilising the pfil(9) framework e.g. dummynet(4), ipfw(8), pf(4), the order in which you load the modules is important. You should kldload the other modules first, as this will ensure TCP packets undergo any necessary manipulations before **SIFTR** "sees" and processes them.

- There is a known, harmless lock order reversal warning between the pfil(9) mutex and tcbinfo TCP lock reported by witness(4) when **SIFTR** is enabled in a kernel compiled with witness(4) support.

- There is no way to filter which TCP flows you wish to capture data for. Post processing is required to separate out data belonging to particular flows of interest.

- The module does not detect deletion of the log file path. New log messages will simply be lost if the log file being used by **SIFTR** is deleted whilst the module is set to use the file. Switching to a new log file using the *net.inet.siftr.logfile* variable will create the new file and allow log messages to begin being written to disk again. The new log file path must differ from the path to the deleted file.

- The hash table used within the code is sized to hold 65536 flows. This is not a hard limit, because chaining is used to handle collisions within the hash table structure. However, we suspect (based on analogies with other hash table performance data) that the hash table look up performance (and therefore the module’s packet processing performance) will degrade in an exponential manner as the number of unique flows handled in a module enable/disable cycle approaches and surpasses 65536.

- There is no garbage collection performed on the flow hash table. The only way currently to flush it is to disable **SIFTR**.

- The PPL variable applies to packets that make it into the processing thread, not total packets received in the hook function. Packets are skipped before the PPL variable is applied, which means there may be a slight discrepancy in the triggering of log messages. For example, if PPL was set to 10, and the 8th packet since the last log message is skipped, the 11th packet will actually trigger the log message to be generated. This is discussed in greater depth in CAIA technical report 070824A.

- At the time of writing, there was no simple way to hook into the TCP layer to intercept packets. **SIFTR**'s use of IP layer hook points means all IP traffic will be processed by the **SIFTR** pfil(9) hook function, which introduces minor, but nonetheless unnecessary packet delay and processing overhead on the system for non-TCP packets as well. Hooking in at the IP layer is also not ideal from the data gathering point of view. Packets traversing up the stack will be intercepted and cause a log message generation BEFORE they have been processed by the TCP layer, which means we cannot observe the cause-and-affect relationship between inbound events and the corresponding TCP control block as precisely as could be. Ideally, **SIFTR** should intercept packets after they have been processed by the TCP layer i.e. intercept packets coming up the stack after they have been processed by **tcp_input**(), and intercept packets coming down the stack after they have been processed by **tcp_output**(). The current code still gives satisfactory granularity though, as inbound events tend to trigger outbound events, allowing the cause-and-effect to be observed indirectly by capturing the state on outbound events as well.

- The "inflight bytes" value logged by **SIFTR** does not take into account bytes that have been SACK'ed by the receiving host.