PATHspider Documentation

Release 1.0.1

the pathspider authors

November 04, 2016

Contents

In today's Internet we see an increasing deployment of middleboxes. While middleboxes provide in-network functionality that is necessary to keep networks manageable and economically viable, any packet mangling — whether essential for the needed functionality or accidental as an unwanted side effect — makes it more and more difficult to deploy new protocols or extensions of existing protocols.

For the evolution of the protocol stack, it is important to know which network impairments exist and potentially need to be worked around. While classical network measurement tools are often focused on absolute performance values, PATHspider performs A/B testing between two different protocols or different protocol extensions to perform controlled experiments of protocol-dependent connectivity problems as well as differential treatment.

PATHspider is a framework for performing and analyzing these measurements, while the actual A/B test can be easily customized. This documentation describes the architecture of PATHspider, the plugins available and how to use and develop the plugins.

CHAPTER 1

Table of Contents

1.1 Introduction

Network operators increasingly rely on in-network functionality to make their networks manageable and economically viable. These middleboxes make the end-to-end path for traffic more opaque by making assumptions about the traffic passing through them. This has led to an ossification of the Internet protocol stack: new protocols and extensions can be difficult to deploy when middleboxes do not understand them *[Honda11]*. PATHspider is a software measurement tool for active measurement of Internet path transparency to transport protocols and transport protocol extensions, that can generate raw data at scale to determine the size and shape of this problem.

The A/B testing measurement methodology used by PATHspider is simple: We perform connections from a set of observation points to a set of measurement targets using two configurations. A baseline configuration (A), usually a TCP connection using kernel default and no extensions, tests basic connectivity. These connections are compared to the experimental configuration (B), which uses a different transport protocol or set of TCP extensions. These connections are made as simultaneously as possible, to reduce the impact of transient network changes.

PATHspider is a generalized version of the ecnspider tool, used in previous studies to probe the paths from multiple vantage points to web-servers *[Trammell15]* and to peer-to-peer clients *[Gubser15]* for failures negotiating Explicit Congestion Notification (ECN) *[RFC3186]* in TCP.

As a generalized tool for controlled experimental A/B testing of path impairment, PATHspider fills a gap in the existing Internet active measurement software ecosystem. Existing active measurement platforms, such as RIPE Atlas *[RIPEAtlas]*, OONI *[Filasto12]*, or Netalyzr *[Kreibich10]*, were built to measure absolute performance and connectivity between a pair of endpoints under certain conditions. The results obtainable from each of these can of course be compared to each other to simulate A/B testing. However, the measurement data obtained from these platforms provide a less controlled view than can be achieved with PATHspider, given coarser scheduling of measurements in each state.

Given PATHspider's modular design and implementation in Python, plugins to perform measurements for any transport protocol or extension are easy to build and can take advantage of the rich Python library ecosystem, including high-level application libraries, low-level socket interfaces, and packet forging tools such as Scapy.

1.1.1 Architecture

The PATHspider architecture has four components, illustrated in the diagram below the *`configurator`*, the `workers`, the *`observer`* and the `merger`. Each component is implemented as one or more threads, launched when PATHspider starts.

For each target hostname and/or address, with port numbers where appropriate, PATHspider enqueues a job, to be distributed amongst the worker threads when available. Each worker performs one connection with the "A" configuration and one connection with the "B" configuration. The "A" configuration will always be connected first and serves

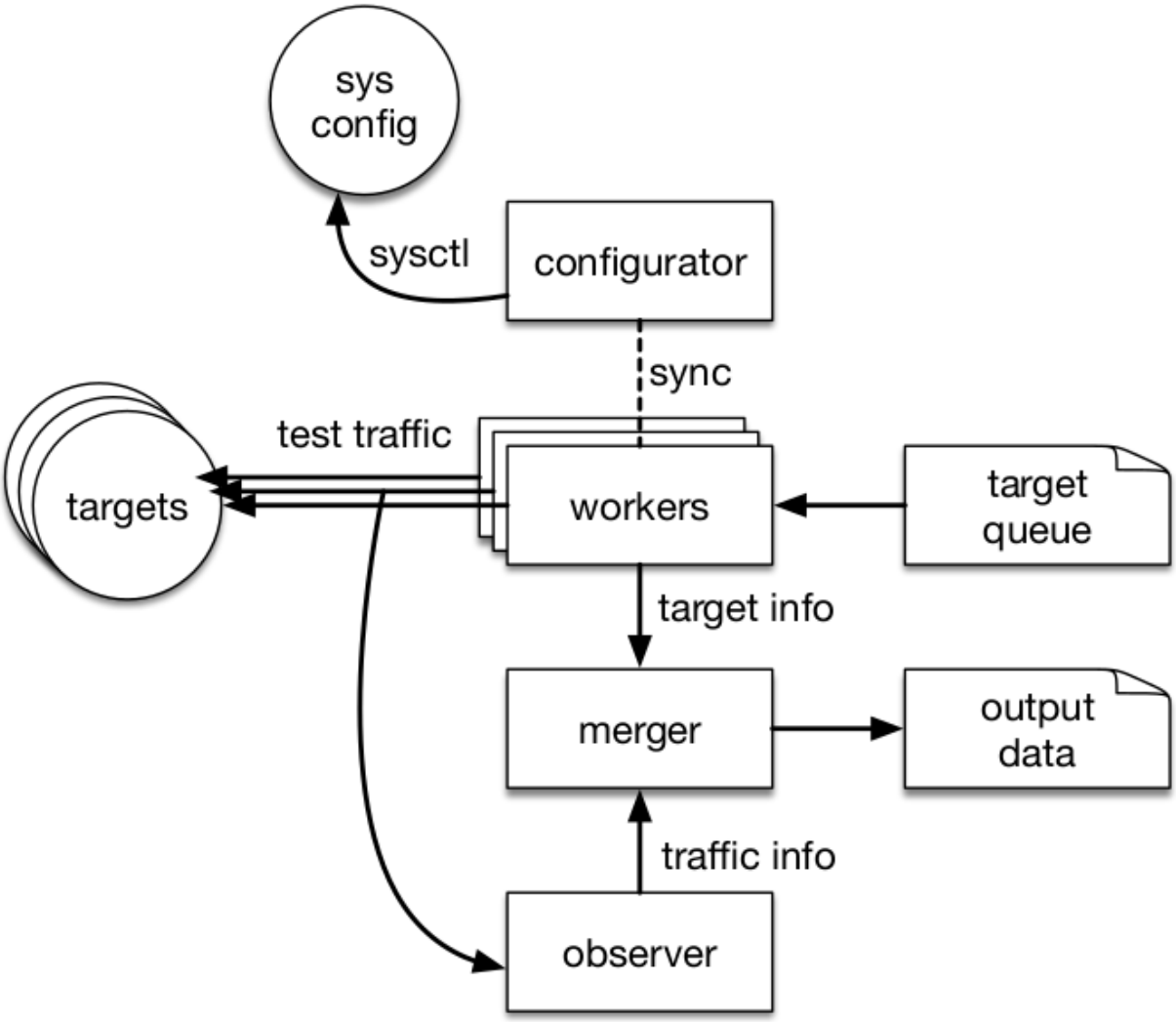

Fig. 1.1: An overview of the PATHspider architecture

as the base line measurement, followed by the "B" configuration. This allows detection of hosts that do not respond rather than failing as a result of using a particular transport protocol or extension. These sockets remain open for a post-connection operation.

Some transport options require a system-wide parameter change, for example enabling ECN in the Linux kernel. This requires locking and synchronisation. Using semaphores, the configurator waits for each worker to complete an operation and then changes the state to perform the next batch of operations. This process cycles continually until no more jobs remain. In a typical experiment, multiple workers (on the order of hundreds) are active, since much of the time in a connection test is spent waiting for an answer from the target or a timeout to fire.

In addition, packets are separately captured for analysis by the observer using Python bindings for libtrace. First, the observer assigns each incoming packet to a flow based on the source and destination addresses, as well as the TCP, UDP or SCTP ports when available. The packet and its associated flow are then passed to a function chain. The functions in this chain may be simple functions, such as counting the number of packets or octets seen for a flow, or more complex functions, such as recording the state of flags within packets and analysis based on previously observed packets in the flow. For example, a function may record both an ECN negotiation attempt and whether the host successfully negotiated use of ECN.

A function may alert the observer that a flow should have completed and that the flow information can be matched with the corresponding job record and passed to the merger. The merger extracts the fields needed for a particular measurement campaign from the records produced by the worker and the observer.

1.1.2 Extensibility

PATHspider plugins are built by extending an abstract class that implements the core behaviour, with functions for the configurator, workers, observer, and matcher.

There are two configurator functions: `config_zero` and `config_one`, run by the configurator to prepare for each attempted connection mode. Where system-wide configuration is not required, the configurator provides the semaphore-based locking functions. This makes the workers aware of the current configuration allowing the connection functions to change based on the current configuration mode.

There are three connection functions: `pre_connect`, `connect` and `post_connect`. `connect` is the only required function. The call to this function is synchronised by the configurator. The `pre_connect` and `post_connect` functions can preconfigure state and perform actions with the connections opened by the `connect` function without being synchronised by the configurator. This can help to speed-up release of the semaphores and complete jobs more efficiently. These actions can also perform data gathering functions, for example, a traceroute to the host being tested.

Plugins can implement arbitrary functions for the observer function chain. These track the state of flows and build flow records for different packet classes: The first chain handles setup on the first packet of a new flow. Separate chains chains for IP, TCP and UDP packets to allow different behaviours based on the IP version and transport protocol.

The final plugin function is the merger function. This takes a job record from a worker and a flow record from the observer and merges the records before passing the merged record back to PATHspider.

1.2 Installation

1.2.1 Debian GNU/Linux

PATHspider is packaged for Debian and packages are made available for the testing and stable-backports distributions. If you are running Debian stable, ensure that you have enabled the stable-backports repository in your apt sources.

To install PATHspider, simply run:

```
sudo apt install pathspider
```

1.2.2 Source

If you are working from the source distribution (e.g. cloned git repository) then you will need to install the required dependencies. On Debian GNU/Linux, assuming you have the stable-backports repository enabled if you are running stable:

```
sudo apt build-dep pathspider
```

Note: This will install both the runtime and the build dependencies required for PATHspider, its testsuite and its documentation.

On other platforms, you may install the dependencies required via pip:

```
pip install -r requirements.txt
```

If you wish to build the documentation from source or to use the testsuite, and you are installing your dependencies via pip, you will also need the following dependencies:

```
pip install -r requirements_dev.txt
```

With the dependencies installed, you can install PATHspider with:

```
python3 setup.py install
```

1.3 Usage Overview

You can run PATHspider from the command line. In order for the Observer to work, you will need permissions to capture raw packets from the network interface. This will require you to use `sudo` or equivalent in order to run PATHspider.

```
# pathspider --help
usage: pathspider [-h] [-s] [-i INTERFACE] [-w WORKERS] [--input INPUTFILE]
                  [--output OUTPUTFILE] [-v]
                  PLUGIN ...

Pathspider will spider the paths.

optional arguments:
  -h, --help            show this help message and exit
  -s, --standalone      run in standalone mode. this is the default mode (and
                        currently the only supported mode). in the future,
                        mplane will be supported as a mode of operation.
  -i INTERFACE, --interface INTERFACE
                        the interface to use for the observer
  -w WORKERS, --workers WORKERS
                        number of workers to use
  --input INPUTFILE     a file containing a list of remote hosts to test, with
                        any accompanying metadata expected by the pathspider
                        test. this file should be formatted as a comma-
                        seperated values file. Defaults to standard input.
  --output OUTPUTFILE   the file to output results data to. Defaults to
```

```
                        standard output.
  -v, --verbose         log debug-level output.

Plugins:
  The following plugins are available for use:

    dscp                DiffServ Codepoints
    tls                 Transport Layer Security
    tfo                 TCP Fast Open
    ecn                 Explicit Congestion Notification
    dnsresolv           DNS resolution for hostnames to IPv4 and v6 addresses

Spider safely!
```

1.3.1 Quickstart Example

You can run a small study using the ECN plugin and the included `webinput.csv` file to measure path transparency to ECN for a small selection of web servers and save the results in `results.txt`:

```
pathspider -i eth0 ecn </usr/share/doc/pathspider/examples/webinput.csv >results.txt
```

Note: If you've not installed PATHspider from apt, you will find the webinput.csv example script in the examples folder of the source distribution.

1.3.2 Data Formats

PATHspider uses newline delimited JSON (ndjson) for the output format. At present, the input format is CSV although in future versions we will deprecate the CSV input format and use a ndjson format input to unify the data formats. The ndjson format gives flexibility in the actual contents of the data as different tests may require data to remain associated with jobs, for example the Alexa ranking of a webserver, so that it can be present in the final output, or in some cases the data may be used as part of the test, for example when running tests against authoritative DNS servers and needing to know a domain for which the server should be authoritative.

Job List

The standalone runner expects a CSV file as input, with one line per job. The format for each line should be as follows:

```
target_ip,target_port,target_hostname,target_rank
```

The current input format is optimised for the use case of using the Alexa top 1 million webservers and so includes a value for the ranking in that list for the job. This value is opaque to PATHspider and may be set to any string desirable, or to `0` if this is not required.

If the `target_port` is not a valid integer, the job will be skipped and a warning emitted by the logger. Blank lines are permitted and will be ignored by the job feeder.

Output Format

PATHspider's output is in the form of two records per job, as JSON dicts. One record will be for the baseline (A) connection, and one for the experimental (B) connection. These JSON records contain the original job information, any information added by the connection functions and any information added by the Observer.

The connection logic of all the plugins that ship with the PATHspider distribution will set a `config` value, either 0 or 1 (with 0 being baseline, 1 being experimental) to distinguish flows. Due to the highly parallel nature of PATHspider, the two flows for a particular job may not be output together and may have other flows between them. Any analysis tools will need to take this into consideration.

The plugins that ship with the PATHspider distribution will also have the following values set in their output:

Key	Description
config	0 for baseline, 1 for experimental
connstate	True if the connection was successful, False if the connection failed (e.g. due to timeout).
dip	Layer 3 (IPv4/IPv6) source address
sp	Layer 4 (TCP/UDP) source port
dp	Layer 4 (TCP/UDP) destination port
pkt_fwd	A count of the number of packets seen in the forward direction
pkt_rev	A count of the number of packets seen in the reverse direction
oct_fwd	A count of the number of octets seen in the forward direction
oct_rev	A count of the number of octets seen in the reverse direction

For detail on the values in individual plugins, see the section for that plugin later in this documentation.

1.4 Plugins

A number of plugins ship with the PATHspider distribution. You can find documentation for them here:

1.4.1 DSCP Plugin

Differentiated services or DiffServ *[RFC2474]* is a networking architecture that specifies a simple, scalable and coarse-grained mechanism for classifying and managing network traffic and providing quality of service (QoS) on modern IP networks. DiffServ can, for example, be used to provide low-latency to critical network traffic such as voice or streaming media while providing simple best-effort service to non-critical services such as web traffic or file transfers.

DiffServ uses a 6-bit differentiated services code point (DSCP) in the 8-bit differentiated services field (DS field) in the IP header for packet classification purposes. The DS field and ECN field replace the outdated IPv4 TOS field. *[RFC3260]*

The DSCP plugin for PATHspider aims to detect breakage in the Internet due to the use of a non-zero DSCP codepoint.

Usage Example

To use the DSCP plugin, specify `dscp` as the plugin to use on the command-line:

```
pathspider dscp </usr/share/doc/pathspider/examples/webtest.csv >results.txt
```

This will run two TCP connections for each job input, one with the DSCP set to zero (best-effort) and one with the DSCP set to 46 (expedited forwarding). If you would like to specify the code point for use on the experimental flow, you may do this with the `--codepoint` option. For example, to use 42:

```
pathspider dscp --codepoint 42 </usr/share/doc/pathspider/examples/webtest.csv >results.txt
```

Output Fields

In addition to the *default output fields*, the DSCP plugin also provides the following fields for each flow:

Key	Description
fwd_syn_dscp	DiffServ code point as observed on the forward path for the first SYN in the flow.
rev_syn_dscp	DiffServ code point as observed on the reverse path for the first SYN in the flow (likely to be a SYN/ACK).
fwd_data_dscp	DiffServ code point as observed on the forward path for the first data packet (i.e. with a payload) in the flow.
rev_data_dscp	DiffServ code point as observed on the reverse path for the first data packet (i.e. with a payload) in the flow.

Notes

- DSCP marking is performed using the `mangle` table in `iptables`. The `config_zero` function will flush this table. PATHspider makes no guarantees the the configuration state is consistent once it has been set, though you can use the forward path markings in the output to validate the results within a reasonably high level of certainty that everything behaved correctly.

1.4.2 ECN Plugin

Explicit Congestion Notification (ECN) is an extension to the Internet Protocol and to the Transmission Control Protocol. *[RFC3186]* ECN allows end-to-end notification of network congestion without dropping packets. ECN is an optional feature that may be used between two ECN-enabled endpoints when the underlying network infrastructure also supports it.

Conventionally, TCP/IP networks signal congestion by dropping packets. When ECN is successfully negotiated, an ECN-aware router may set a mark in the IP header instead of dropping a packet in order to signal impending congestion. The receiver of the packet echoes the congestion indication to the sender, which reduces its transmission rate as if it detected a dropped packet.

Rather than responding properly or ignoring the bits, some outdated or faulty network equipment has historically dropped or mangled packets that have ECN bits set. As of 2015, measurements suggested that the fraction of web servers on the public Internet for which setting ECN prevents network connections had been reduced to less than 1%. *[Trammell15]*

The ECN plugin for PATHspider aims to detect breakage in the Internet due to the use of ECN.

Usage Example

To use the ECN plugin, specify `ecn` as the plugin to use on the command-line:

```
pathspider ecn </usr/share/doc/pathspider/examples/webtest.csv >results.txt
```

This will run two TCP connections for each job input, one with ECN disabled in the kernel TCP/IP stack and one with ECN enabled in the kernel TCP/IP stack.

Output Fields

In addition to the *default output fields*, the ECN plugin also provides the following fields for each flow:

Key	Description
fwd_ez	ECT(0) was observed in the forward direction.
rev_ez	ECT(0) was observed in the reverse direction.
fwd_eo	ECT(1) was observed in the forward direction.
rev_eo	ECT(1) was observed in the reverse direction.
fwd_ce	CE was observed in the forward direction.
rev_ce	CE was observed in the reverse direction.
fwd_syn_flags	The SYN flags observed in the forward direction.
rev_syn_flags	The SYN flags observed in the reverse direction.
fwd_fin	A FIN flag was observed in the forward direction.
rev_fin	A FIN flag was observed in the reverse direction.
fwd_rst	A RST flag was observed in the forward direction.
rev_rst	A RST flag was observed in the reverse direction.
tcp_completed	A complete 3WHS for TCP was observed to be successful.

Notes

- ECN behaviour is implemented by the host kernel for PATHspider, and is switched by a `sysctl` call. PATHspider makes no guarantees the the configuration state is consistent once it has been set, though you can use the forward SYN flags in the output to validate the results within a reasonably high level of certainty that everything behaved correctly.

1.4.3 TFO Plugin

TCP Fast Open (TFO) is an extension to speed up the opening of successive Transmission Control Protocol (TCP) connections between two endpoints. It works by using a TFO cookie (a TCP option), which is a cryptographic cookie stored on the client and set upon the initial connection with the server. *[RFC7413]*

When the client later reconnects, it sends the initial SYN packet along with the TFO cookie data to authenticate itself. If successful, the server may start sending data to the client even before the reception of the final ACK packet of the three-way handshake, skipping that way a round-trip delay and lowering the latency in the start of data transmission.

The TFO plugin for PATHspider aims to detect connectivity breakage due to the the use of TCP Fast Open, implementation of TCP Fast Open, and TFO implementation anomalies.

Usage Example

To use the TFO plugin, specify `tfo` as the plugin to use on the command-line:

```
pathspider tfo </usr/share/doc/pathspider/examples/webtest.csv >results.txt
```

For the baseline test, the plugin will perform a TCP connection to the target host. For the experimental case, the plugin will perform two TCP connections to the target host. The first experimental connection is run to acquire a TFO cookie and the second to check that it can be used.

Output Fields

In addition to the *default output fields*, the TFO plugin also provides the following fields for each flow:

Key	Description
tfo_synkind	TCP Option Kind of TFO option on SYN (34, 254; 0 = none)
tfo_ackkind	TCP Option Kind of TFO option on SYN/ACK (34, 254; 0 = none)
tfo_synclen	TFO Cookie Length on SYN
tfo_ackclen	TFO Cookie Length on SYN/ACK
tfo_seq	Sequence number of SYN
tfo_dlen	Length of TCP payload on SYN
tfo_ack	Ack number of SYN/ACK. For ACKed data, = seq + dlen + 1

1.4.4 3rd-Party Plugins

You will be able to list the 3rd-party plugins installed by running:

```
pathspider --help
```

There is no need to register 3rd-party plugins with PATHspider before use, they will be automatically detected once they are installed.

1.5 Using the Resolver

The resolver accepts input formatted as CSV in the style of the Alexa top 1 million website listing:

```
rank,domain
```

The output format is the native input format for PATHspider plugins.

1.5.1 Basic Usage

```
usage: pathspider dnsresolv [-h] [--timeout TIMEOUT] [--sleep SLEEP]
                            [--add-port ADD_PORT]
                            [--www {never,preferred,always,both}]
                            [--debug-skip DEBUG_SKIP]
                            [--debug-count DEBUG_COUNT]

optional arguments:
  -h, --help            show this help message and exit
  --timeout TIMEOUT, -t TIMEOUT
                        Timeout for DNS resolution.
  --sleep SLEEP, -s SLEEP
                        Sleep before every request. Useful for rate-limiting.
  --add-port ADD_PORT, -p ADD_PORT
                        If specified, this port number will be added to
                        everyline in the output file.
  --www {never,preferred,always,both}
                        Mode for prepending "www." to every domain before
                        resolution. "never" will never prepend "www.".
                        "preferred" will prepend "www." if the resolution of
                        the domain including "www." is successful (more
                        specifically: an A record is returned), and otherwise
                        fall back to omitting the "www.". "always" will
                        prepend "www." and will return no IP address in the
                        output file, even when the domain without "www." can
                        be resolved to one. "both" behaves as "always" and
```

```
                        "never" together, that is, it resolves each domain
                        with and without a prepended "www.". All values for
                        this option will never stack the www's, that is
                        "www.example.com" will never be expanded to
                        "www.www.example.com". An existing "www." prefix from
                        a domain from the input file will never be dropped. If
                        this value is not "never", then the output file may
                        contain different FQDNs from the input file, as
                        "example.com" might be turned into "www.example.com".
  --debug-skip DEBUG_SKIP
                        Skip the first N domains, and do not resolve them.
  --debug-count DEBUG_COUNT
                        Perform resolution for at most N domains. All of them
                        if this value is set to 0.
```

1.5.2 Example Usage

```
pathspider dnsresolv <alexa-1m.csv >input-list.txt
```

1.6 Developing Plugins

PATHspider is written to be extensible and the plugins that are included in the PATHspider distribution are examples of the measurements that PATHspider can perform.

`pathspider.plugins` is a namespace package. Namespace packages are a mechanism for splitting a single Python package across multiple directories on disk. One or more distributions may provide modules which exist inside the same namespace package. The PATHspider distribution's plugins are installed in `pathspider.plugins`, but also 3rd-party plugins can exist in this path without being a part of the PATHspider distribution.

1.6.1 Quickstart

The directory layout and example plugin below can be found in the pathspider-example GitHub repository. You can get going quickly by forking this repository and using that as a basis for plugin development.

1.6.2 Directory Layout

To get started you will need to create the required directory layout for PATHspider plugins, in this case for the Example plugin:

```
pathspider-example
-- pathspider
    -- __init__.py
    -- plugins
        -- __init__.py
        -- example.py
```

Inside both __init__.py files, you will need to add the following (and only the following):

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

Your plugin will be written in `example.py` and this plugin will be discovered automatically when you run PATHspider.

1.6.3 Example Plugin

The following code can be found in the quickstart example as a starting point for developing your plugin. If you are not using the quickstart example, you may copy and paste this code into a Python file under `pathspider/plugins/` in the directory structure. This example is explained in the following sections.

```
import sys
import collections
import logging

from pathspider.base import SynchronizedSpider
from pathspider.base import PluggableSpider
from pathspider.base import Conn
from pathspider.base import NO_FLOW

from pathspider.observer import simple_observer

SpiderRecord = collections.namedtuple("SpiderRecord",
        ["ip", "rport", "port", "rank", "host", "config",
        "connstate", "tstart", "tstop"])

class Example(SynchronizedSpider, PluggableSpider):
    """
    An example PATHspider plugin.
    """

    def config_zero(self):
        logger = logging.getLogger("example")
        logger.debug("Configuration zero")

    def config_one(self):
        logger = logging.getLogger("example")
        logger.debug("Configuration one")

    def connect(self, job, pcs, config):
        return self.tcp_connect(job)

    def post_connect(self, job, conn, pcs, config):
        job_ip, job_port, job_host, job_rank = job
        tstop = str(datetime.utcnow())

        if conn.state == Conn.OK:
            rec = SpiderRecord(job_ip, job_port, conn.port, job_rank, job_host,
                               config, True, conn.tstart, tstop)
        else:
            rec = SpiderRecord(job_ip, job_port, conn.port, job_rank, job_host,
                               config, False, conn.tstart, tstop)

        try:
            conn.client.shutdown(socket.SHUT_RDWR)
        except:
            pass

        try:
```

```
            conn.client.close()
        except:
            pass

         return rec

    def create_observer(self):
        logger = logging.getLogger("example")
        try:
            return simple_observer()
        except:
            logger.error("Observer would not start")
            sys.exit(-1)

    def merge(self, flow, res):
        if flow == NO_FLOW:
            flow = {"dip": res.ip,
                    "sp": res.port,
                    "dp": res.rport,
                    "observed": False}
        else:
            flow['observed'] = True

        self.outqueue.put(flow)

    @staticmethod
    def register_args(subparsers):
        parser = subparsers.add_parser('example', help="Example starting point for development")
        parser.set_defaults(spider=Example)
```

You will need to provide implementations for each of these functions, which are explained next. We'll start with the connection logic.

1.6.4 Connection Logic

Configurator

These functions perform global changes that may be required between performing the baseline (A) and the experimental (B) configurations. The changes may be a call to sysctl, changes via netfilter or a call to a robot arm to reposition the satellite array. In the event that global state changes are not required, these can be implemented as no-ops.

An example implementation of these methods can be found in the ECN plugin:

ECN.**config_zero**()
: Disables ECN negotiation via sysctl.

ECN.**config_one**()
: Enables ECN negotiation via sysctl.

(Pre-,Post-) Connection

The pre-connection function will run only once, and the result of the pre-connection operation will be available to both runs of the connection and post-connection functions.

If you require to pass different values depending on the configuration, you can perform two operations in the pre-connect function, returning a tuple, and selecting the value to use based on the configuration in the later functions.

An example implementation of these methods can be found in the ECN plugin:

ECN.**connect**(*job*, *pcs*, *config*)
Performs a TCP connection.

ECN.**post_connect**(*job*, *conn*, *pcs*, *config*)
Close the socket gracefully.

1.6.5 Observer Functions

PATHspider's observer will accept functions and pass python-libtrace dissected packets along with the associated flow record to them for every packet recieved.

The *pathspider.observer* module provides *pathspider.observer.simple_observer()* which allows the creation of a very simple Observer during development of the other portions of the plugin. There are two simple examples of observer functions that are used in the observer created by this function.

When you are ready to start working with your own Observer functions, you will need to expand your `create_observer()` function. You can use the following example:

```
from pathspider.observer import Observer
from pathspider.observer import basic_flow
from pathspider.observer import basic_count

class Example(SynchronizedSpider, PluggableSpider):

    [...]

    def create_observer(self):
        logger = logging.getLogger("example")
        try:
            return Observer(self.libtrace_uri,
                            new_flow_chain=[basic_flow],
                            ip4_chain=[basic_count],
                            ip6_chain=[basic_count])
        except:
            logger.error("Observer would not start")
            sys.exit(-1)
```

Depending on the types of analysis you would like to do on the packets, you should pass your functions to the appropriate chain:

Function Chain	Description
new_flow_chain	Functions to initialise fields in the flow record for new flows.
ip4_chain	Functions to record details from IPv4 headers.
ip6_chain	Functions to record details from IPv6 headers.
tcp_chain	Functions to record details from TCP headers.
udp_chain	Functions to record details from UDP headers.
l4_chain	Functions to record details from other layer 4 headers.

Library Observer Functions

The *pathspider.observer.basic_flow()* function simply creates the inital state for the flow record, extracting the 5-tuple and initialising counters. The counters are used by the *pathspider.observer.basic_count()* function that counts the number of packets and octets seen in each direction. These combined will allow your plugin to produce the *default output fields*.

PATHspider also provides library observer functions for some protocols:

pathspider.observer.icmp

pathspider.observer.icmp.**icmp_setup**(*rec*, *ip*)

pathspider.observer.icmp.**icmp_unreachable**(*rec*, *ip*, *q*, *rev*)

pathspider.observer.tcp

pathspider.observer.tcp.**tcp_complete**(*rec*, *tcp*, *rev*)

pathspider.observer.tcp.**tcp_handshake**(*rec*, *tcp*, *rev*)

pathspider.observer.tcp.**tcp_setup**(*rec*, *ip*)

Writing Observer Functions

When you are ready to write functions for the observer, first identify which data should be stored in the flow record. This is a `dict` that is made available for every call to an observer function for a particular flow and not shared across flows. Once the flow is completed, this is the record that will be returned to the merger.

The flow record should be initialised when a new flow has been identified. The functions in the `new_flow_chain` are called, in sequence, when a new flow is identified by the Observer. These functions are passed two arguments: `rec` - the empty flow record, and `ip` - the IP header.

You should familiarise yourself with the python-libtrace documentation. The analysis functions all follow the same function prototype with `rec` - the empty flow record, `x` - the header, and `rev` - boolean value indicating the direction the packet travelled (i.e. Was the packet in the reverse direction?).

The only difference in these functions is the header that is passed, as a python-libtrace object, to the function. The same flow record is always passed for each call for the same flow, regardless of which function chain the function is in.

If a function returns False, as it has identified the end of the flow, the Observer will consider the flow to be finished and will pass it to be merged with the job record after a short delay. This might occur for TCP flows when both FIN packets have been seen using the *pathspider.observer.tcp.tcp_complete()* function.

1.6.6 Merging

The merge function will be called for every job and given the job record and the observer record. The merge function is then to return the final record to be recorded in the dataset for the measurement run.

> **Warning:** It is possible for the Observer to return a NO_FLOW object in some circumstances, where the flow has not been observed. Any implementation must handle this gracefully.

An example implementation of this method can be found in the ECN plugin:

ECN.**merge**(*flow*, *res*)

Merge flow records.

Includes the configuration and connection success or failure of the socket connection with the flow record.

1.6.7 Running Your Plugin

In order to run your plugin, in the root of your plugin source tree run:

```
PYTHONPATH=. pathspider example </usr/share/doc/pathspider/examples.csv >results.txt
```

Unless you install your plugin, you will need to add the plugin tree to the `PYTHONPATH` to allow the plugin to be discovered.

1.7 Advanced Topics

1.7.1 PATHspider Internals

To learn more about the internals of PATHspider, you can read the following pages describing the operation of individual parts of the architecture:

Abstract Spider

The core functionality of PATHspider is implemented in two classes: `pathspider.base.SyncronisedSpider` and `pathspider.base.DesynchronisedSpider`. These both inherit from the base *`pathspider.base.Spider`* which provides a skeleton that has the required functions for any plugin. The documentation for this base class is below:

pathspider.base

Basic framework for Pathspider: coordinate active measurements on large target lists with both system-level network stack state (sysctls, iptables rules, etc) as well as information derived from flow-level passive observation of traffic at the sender.

Derived and generalized from ECN Spider (c) 2014 Damiano Boppart <hat.guy.repo@gmail.com>

> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

class `pathspider.base.`**`Conn`**

class `pathspider.base.`**`Connection`**(*client*, *port*, *state*, *tstart*)

> **`client`**
> Alias for field number 0
>
> **`port`**
> Alias for field number 1
>
> **`state`**
> Alias for field number 2

tstart
Alias for field number 3

class pathspider.base.**DesynchronizedSpider**(*worker_count*, *libtrace_uri*, *args*)

config_one()

config_zero()

configurator()
Since there is no need for a configurator thread in a desynchronized spider, this thread is a no-op

worker(*worker_number*)
This function provides the logic for configuration-synchronized worker threads.

Parameters **worker_number** (*int*) – The unique number of the worker.

The workers operate as continuous loops:

•Fetch next job from the job queue

•Perform pre-connection operations

•Acquire a lock for "config_zero"

•Perform the "config_zero" connection

•Release "config_zero"

•Acquire a lock for "config_one"

•Perform the "config_one" connection

•Release "config_one"

•Perform post-connection operations for config_zero and pass the result to the merger

•Perform post-connection operations for config_one and pass the result to the merger

•Do it all again

If the job fetched is the SHUTDOWN_SENTINEL, then the worker will terminate as this indicates that all the jobs have now been processed.

class pathspider.base.**PluggableSpider**

static register_args(*subparsers*)

class pathspider.base.**SemaphoreN**(*value*)
An extension to the standard library's BoundedSemaphore that provides functions to handle n tokens at once.

acquire_n(*value=1*, *blocking=True*, *timeout=None*)
Acquire `value` number of tokens at once.

The parameters `blocking` and `timeout` have the same semantics as `BoundedSemaphore`.

Returns The same value as the last call to *BoundedSemaphore*'s

`acquire()` if `acquire()` were called `value` times instead of the call to this method.

empty()
Acquire all tokens of the semaphore.

release_n(*value=1*)
Release `value` number of tokens at once.

Returns The same value as the last call to *BoundedSemaphore*'s `release()` if `release()` were called `value` times instead of the call to this method.

class `pathspider.base.`**Spider**(*worker_count*, *libtrace_uri*, *args*)

A spider consists of a configurator (which alternates between two system configurations), a large number of workers (for performing some network action for each configuration), an Observer which derives information from passively observed traffic, and a thread that merges results from the workers with flow records from the collector.

add_job(*job*)

Adds a job to the job queue.

If PATHspider is currently stopping, the job will not be added to the queue.

config_one()

Changes the global state or system configuration for the experimental measurements.

config_zero()

Changes the global state or system configuration for the baseline measurements.

configurator()

connect(*job*, *pcs*, *config*)

Performs the connection.

Parameters

- **job** (*dict*) – The job record.
- **pcs** (*dict*) – The result of the pre-connection operations(s).
- **config** (*int*) – The current state of the configurator (0 or 1).

Returns object – Any result of the connect operation to be passed to *pathspider.base.Spider.post_connect()*.

The connect function is used to perform the connection operation and is run for both the A and B test. This method is not implemented in the abstract *pathspider.base.Spider* class and must be implemented by any plugin.

Sockets created during this operation can be returned by the function for use in the post-connection phase, to minimise the time that the configurator is blocked from moving to the next configuration.

create_observer()

Create a flow observer.

This function is called by the base Spider logic to get an instance of *pathspider.observer.Observer* configured with the function chains that are requried by the plugin.

This method is not implemented in the abstract *pathspider.base.Spider* class and must be implemented by any plugin.

For more information on how to use the flow observer, see *Observer*.

exception_wrapper(*target*, **args*, ***kwargs*)

merge(*flow*, *res*)

Merge a job record with a flow record.

Parameters

- **flow** (*dict*) – The flow record.
- **res** (*dict*) – The job record.

Returns tuple – Final record for job.

In order to create a final record for reporting on a job, the final job record must be merged with the flow record. This function should be implemented by any plugin to provide the logic for this merge as the keys used in these records cannot be known by PATHspider in advance.

This method is not implemented in the abstract *pathspider.base.Spider* class and must be implemented by any plugin.

merger()
Thread to merge results from the workers and the observer.

post_connect(*job*, *conn*, *pcs*, *config*)
Performs post-connection operations.

Parameters

- **job** (*dict*) – The job record.
- **conn** (*object*) – The result of the connection operation(s).
- **pcs** (*dict*) – The result of the pre-connection operations(s).
- **config** (*int*) – The state of the configurator during *pathspider.base.Spider.connect()*.

Returns dict – Result of the pre-connection operation(s).

The post_connect function can be used to perform any operations that must be performed after each connection. It will be run for both the A and the B configuration, and is not synchronised with the configurator.

Plugins to PATHspider can optionally implement this function. If this function is not overloaded, it will be a noop.

Any sockets or other file handles that were opened during *pathspider.base.Spider.connect()* should be closed in this function if they have not been already.

pre_connect(*job*)
Performs pre-connection operations.

Parameters **job** (*dict*) – The job record.

Returns dict – Result of the pre-connection operation(s).

The pre_connect function can be used to perform any operations that must be performed before each connection. It will be run only once per job, with the same result passed to both the A and B connect calls. This function is not synchronised with the configurator.

Plugins to PATHspider can optionally implement this function. If this function is not overloaded, it will be a noop.

shutdown()
Shut down PathSpider in an orderly fashion, ensuring that all queued jobs complete, and all available results are merged.

start()
This function starts a PATHspider plugin.

In order to run, the plugin must have first been activated by calling its `activate()` method. This function causes the following to happen:

•Set the running flag

•Create an *pathspider.observer.Observer* and start its process

•Start the merger thread

- Start the configurator thread
- Start the worker threads

The number of worker threads to start was given when activating the plugin.

terminate()
Shut down PathSpider as quickly as possible, without any regard to completeness of results.

worker()

class pathspider.base.**SynchronizedSpider**(*worker_count*, *libtrace_uri*, *args*)

configurator()
Thread which synchronizes on a set of semaphores and alternates between two system states.

tcp_connect(*job*)
This helper function will perform a TCP connection. It will not perform any special action in the event that this is the experimental flow, it only performs a TCP connection. This function expects that self.conn_timeout has been set to a sensible value.

worker(*worker_number*)
This function provides the logic for configuration-synchronized worker threads.

Parameters **worker_number** (*int*) – The unique number of the worker.

The workers operate as continuous loops:

- Fetch next job from the job queue
- Perform pre-connection operations
- Acquire a lock for "config_zero"
- Perform the "config_zero" connection
- Release "config_zero"
- Acquire a lock for "config_one"
- Perform the "config_one" connection
- Release "config_one"
- Perform post-connection operations for config_zero and pass the result to the merger
- Perform post-connection operations for config_one and pass the result to the merger
- Do it all again

If the job fetched is the SHUTDOWN_SENTINEL, then the worker will terminate as this indicates that all the jobs have now been processed.

Observer

class pathspider.observer.**Observer**(*lturi*, *new_flow_chain=[]*, *ip4_chain=[]*, *ip6_chain=[]*, *icmp4_chain=[]*, *icmp6_chain=[]*, *tcp_chain=[]*, *udp_chain=[]*, *l4_chain=[]*, *idle_timeout=30*, *expiry_timeout=5*)
Wraps a packet source identified by a libtrace URI, parses packets to divide them into flows, passing these packets and flows onto a function chain to allow data to be associated with each flow.

__init__(*lturi*, *new_flow_chain=[]*, *ip4_chain=[]*, *ip6_chain=[]*, *icmp4_chain=[]*, *icmp6_chain=[]*, *tcp_chain=[]*, *udp_chain=[]*, *l4_chain=[]*, *idle_timeout=30*, *expiry_timeout=5*)

Create an Observer.

Parameters

- **new_flow_chain** (*array(function)*) – Array of functions to initialise new flows.
- **ip4_chain** (*array(function)*) – Array of functions to pass IPv4 headers to.
- **ip6_chain** (*array(function)*) – Array of functions to pass IPv6 headers to.
- **icmp4_chain** (*array(function)*) – Array of functions to pass IPv4 headers containing ICMPv4 headers to.
- **icmp6_chain** (*array(function)*) – Array of functions to pass IPv6 headers containing ICMPv6 headers to.
- **tcp_chain** (*array(function)*) – Array of functions to pass TCP headers to.
- **tcp_chain** – Array of functions to pass TCP headers to.
- **udp_chain** (*array(function)*) – Array of functions to pass UDP headers to.
- **l4_chain** (*array(function)*) – Array of functions to pass other layer 4 headers to.

See also *Observer Documentation*

flush()

run_flow_enqueuer(*flowqueue*, *irqueue=None*)

class pathspider.observer.**PacketClockTimer**(*time*, *fn*)

fn
Alias for field number 1

time
Alias for field number 0

pathspider.observer.**basic_count**(*rec*, *ip*, *rev*)
Packet function that counts packets and octets per flow

pathspider.observer.**basic_flow**(*rec*, *ip*)
New flow function that sets up basic flow information

pathspider.observer.**extract_ports**(*ip*)

pathspider.observer.**simple_observer**(*lturi*)

1.7.2 PATHspider on Vagrant

On systems other than Linux systems, you may use Vagrant to run PATHspider. This may also be useful during development. A Vagrantfile is provided that will create a Debian-based virtual machine with all the PATHspider dependencies installed.

In the virtual machine, the PATHspider code will be mounted at */home/vagrant/pathspider* and changes made inside or outside the VM will appear in both places. PATHspider is installed in development mode, meaning that this is also the location of the PATHspider code that will be run when running the */usr/bin/pathspider* binary inside the virtual machine.

1.7.3 PATHspider on MONROE

PATHspider provides a Docker container that may be extended by experimenters using the MONROE testbed. You can read more about how to use PATHspider on MONROE in the project's README.

1.8 References

CHAPTER 2

Citing PATHspider

When presenting work that uses PATHspider, we would appreciate it if you could cite PATHspider as:

> Learmonth, I.R., Trammell, B., Kuhlewind, M. and Fairhurst, G., 2016, July. PATHspider: A tool for active measurement of path transparency. In Proceedings of the 2016 Applied Networking Research Workshop (pp. 62-64). ACM.

CHAPTER 3

Acknowledgements

Current development of PATHspider is supported by the European Union's Horizon 2020 project MAMI. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688421. The opinions expressed and arguments employed reflect only the authors' view. The European Commission is not responsible for any use that may be made of that information.

Bibliography

[Honda11] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M. and Tokuda, H., 11, November. Is it still possible to extend TCP?. In Proceedings of the 11 ACM SIGCOMM conference on Internet measurement conference (pp. 181-194). ACM.

[Trammell15] Trammell, B., Kühlewind, M., Boppart, D., Learmonth, I., Fairhurst, G. and Scheffenegger, R., 15, March. Enabling Internet-wide deployment of explicit congestion notification. In International Conference on Passive and Active Network Measurement (pp. 193-205). Springer International Publishing.

[Gubser15] Gubser, E., Measuring Explicit Congestion Negotiation (ECN) support based on P2P networks, 2015.

[RFC2474] Nichols, K., Blake, S., Baker, F. and Black, D., 1998. **Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers**. RFC Editor.

[RFC3186] Ramakrishnan, K., Floyd, S. and Black, D., 2001. **The addition of explicit congestion notification (ECN) to IP**. RFC Editor.

[RFC3260] Grossman, D., 2002. **New terminology and clarifications for DiffServ**. RFC Editor.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S. and Jain, A., 2014. **TCP Fast Open**. RFC Editor.

[RIPEAtlas] Ripe, N.C.C.. RIPE atlas.

[Filasto12] Filasto, A. and Appelbaum, J., 2012, August. OONI: Open Observatory of Network Interference. In FOCI.

[Kreibich10] Kreibich, C., Weaver, N., Nechaev, B. and Paxson, V., 2010, November. Netalyzr: illuminating the edge network. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (pp. 246-259). ACM.

p

Symbols

A

B

C

D

E

F

I

M

O

P

R

S

T

W