This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 688421.

Measurement and Architecture for a Middleboxed Internet

H2020-ICT-688421

Middlebox Observatory Design and Data Model

Author(s):	ETH	B. Trammell (ed.)
	UNIABDN	Iain R. Learmonth
	ZHAW	Stephan Neuhaus

Document Number:	D1.2
Internal Reviewer:	Diego Lopez
Due Date of Delivery:	30 June 2018
Actual Date of Delivery:	30 June 2018
Dissemination Level:	Public

Disclaimer

Contents

European Commission | Horizon 2020 European Union funding for Research & Innovation

Executive Summary

This deliverable describes the design principles, architecture, and information and data models for the MAMI Path Transparency Observatory (PTO; referred to as the Middlebox Observatory in our description of work and earlier deliverables) and the PATHspider measurement tool, which is its primary source of information.

The PTO is based around a simple API which places measurement metadata at the core of the collection and analysis workflow. It keeps measurement data in raw form, but provides access to normalized and aggregated data in a simple observation data model, supporting comparability of measurements taken with different tools. The emphasis on metadata supports the retrieval of provenance information for all queries and observations, which supports repeatability of experiments, and the normalization step supports data reduction, e.g., to remove unnecessary detail about network topology or IP addresses linked to users.

This design represents a refinement on earlier iterations of the PTO detailed in our earlier deliverable D1.1 and in publications by the project in 2016 and 2017.

The content of this document is largely taken from an academic paper currently under submission, and from the documentation for the PTO.

1 Introduction

The science of Internet measurement relies on diverse kinds of analysis, applied to data collected from various sources using many different techniques. Key to extracting insight and knowledge from these efforts, and to supporting repeatability and comparability of results, is the management of data and analyses as well as metadata at multiple resolutions. These insights are in no way new [15], but the continually growing scale of the Internet has increased the need for less ad-hoc approaches to data and metadata management. At the same time, the Internet measurement community has devoted increasing attention to the impact of measurement data collection on end-user privacy. The tension between maximizing data utility and minimizing risk complicates the development of such approaches.

A key deliverable of the MAMI project is the Path Transparency Observatory (termed Middlebox Observatory in our proposal and description of work), an Internet measurement *Observatory* focused on path teansparency measurement. Refining the notion of Internet transparency [1], we call a path transparent to a given protocol feature if packets travel unaltered from one end of the path to the other, independent of the use (or attempted use) of that protocol feature. Measurements of path transparency can be used to infer the presence, topological location, and effects of middleboxes, as well as end-host support for these features.

Examples of impairments to transparency include the stripping of TCP options [11], modification of the former IP Type of Service field [8], or blocking of UDP traffic [7], which can additionally impair UDP-encapsulated protocols like QUIC [20]. Such impairments are either intentional or unintentional side-effect from middleboxes but increasingly limit our ability to deploy new protocols. Solutions to work around this ossification should be based on empirical data about which impairments can be observed in the Internet and their prevalence. However, today's engineering decisions, which can significantly increase protocol complexity, are often based on single measurement studies (c.f. HICCUPS [5] and Multipath TCP [17]). Therefore it is especially important for the case of path transparency to collect and manage large-scale and longitudinal data to understand the observed behavior. Data collection in the Path Transparency Observatory (PTO) supports finding quantitative insights on the extent and nature of this ossification, and further enable us to monitor changes over time.

The basic concepts implemented by the PTO can be applied to most Internet measurement tasks, active or passive. Only the information model is task-specific, and in our project is focused on providing answers to questions about the nature and prevalence of path impairments in the Internet. However, the key feature of our approach is the separation of raw *measurement data*, from diverse sources, from *observations* that are generated by an intermediate analysis step. The former is kept in its original form and annotated with metadata about how it can be transformed into the latter. This transformation step, called *normalization*, achieves both data reduction and comparability. Data reduction reduces inadvertent risk to end-user privacy, and comparability means that observations are comparable, even across different campaigns and across different instances of the observatory, as long as these instances use the same information model. Further metadata about queries, observations, and raw data is designed to allow full discovery of the *provenance* of any particular observation, supporting repeatability of result generation independent of the current state of the observatory.

The PTO's design principles and architecture are described in Chapter 2, and its information model in Chapter 3. Chapter 4 describes the API provided by the observatory, and Chapter 5 describes the interface presented to analyzers and normalizers that generate observations from raw data. Finally, Chapter 6 discusses our implementation of these interfaces based on this architecture.

This document is largely derived from the documentation of the PTO, available online at

https://github.com/mami-project/pto3-go

and from an academic paper detailing the PTO under submission at the time of writing of this deliverable. It is presented in this form as a more convenient extract than references to these sources.

Horizon 2020
European Union funding
for Research & Innovation

2 PTO Architecture and Concepts

The architecture of the PTO has three main goals in collecting large-scale data and making them available for research, protocol engineering, and operation: maximizing comparability of data across diverse measurement campaigns, formats and tools; supporting repeatability of experimentation; and providing protection of raw measurement data. We discuss these design principles in Section 2.1.

Since metadata is key to understanding and maintaining research data for comparability and repeatability of experiments, the workflow of the PTO is centered around metadata. It is used to keep track of the provenance of objects, to control the operation in the analysis stage, and to control the operation of the PTO itself when uploading or querying data.

The PTO is built around a RESTful interface (defined in detail in Chapter 4) with three types of resources (see Figure 1): *raw data* generated directly by measurement tools in diverse data formats and the metadata describing it; *observation* data and the metadata describing it, *normalized* from the raw data into a common information model, or derived from the *analysis* of other observations; and *queries* over the set of observation data, including metadata describing the queries, providing results to the user.

Data and metadata objects are stored as separate resources: while raw data, observations, and query results are immutable, the metadata describing these may be updated, and updates to metadata are used to control most actions in the PTO.

2.1 Design Principles

We take these as design principles, leading to an architecture which is actually independent of the type of measurement data stored in the observatory; the fitness of our observatory for the measurements undertaken by the project, those of Internet path transparency, are dictated by the information model for observations we detail in Chapter 3 and the details of the query API detailed in Section 4.3.

We discuss these design principles in more detail in the sections below, before exploring the design of the observatory in depth.

2.1.1 Comparability

The difficulty of building and maintaining a data collection infrastructure or arranging access to Internet measurement data leads to a tendency to “look under lampposts”: a given measurement campaign tends to focus more on deriving insight from what it finds easy to measure, rather than measuring where maximum insight is available. However, everyone has different lampposts, and adding vantage points may provide useful new information. Combining data from all these vantage points requires data to be comparable. This leads us to separate raw data from normalized observations, which are independent of the form of raw data. It is these observations, and analyses based upon them, that the PTO presents to its users. The details of this information model, and how it supports this comparability, are given in Chapter 3.

2.1.2 Repeatability

Every observation in the PTO is backed by raw data. All raw data and observations carry freely defined metadata, and the PTO tracks the provenance of every observation and query made, back to the raw data. The normalizers and analyzers that generate the observations are also associated with metadata including a public source repository reference, and raw metadata can also refer back to the tools and configurations used to run the original measurements. A user of the PTO therefore has all the information necessary to repeat a measurement, which can aid reproducibility, the relative lack of which in our field is a topic of current concern [19].

In addition, the PTO is designed as a write-only, stable reference for data sets. Deletion and deprecation of raw data and observations is supported as an exception, generally during the development and testing of an analyzer, or to handle later discovery of methodological flaws in a measurement.

2.1.3 Protection

Even though active measurement campaigns, run from purpose-built infrastructure, present fewer data protection challenges than passive measurement campaigns, the PTO protects raw measurement data in multiple ways. First, all raw measurement data is associated with an owner, and can only be accessed by the owner or by normalizers which are authorized (and in some cases developed and maintained) by the owner. Second, normalizers have the effect of reducing the data, and therefore the risk it represents, to a common data model, designed to filter out irrelevant information. Third, normalizers are run manually, and the workflow allows review both of normalizer code and of output, in order to further reduce the risk presented in raw data access.

2.2 Data flow

Every analysis begins with raw data. Raw data is stored as plain files directly in the format generated by the tools performing the measurements. Some tools may be designed specifically to work with the observatory, such as PATHspider [22], which outputs newline-delimited JSON containing grouped summarized flow data that can be trivially normalized into observations. Data from other tools may need more significant normalization, e.g. IPFIX flow data [10] for passive measurement of UDP impairments [7]. Since many measurement tasks produce multiple files related to a single measurement, raw data is organized into *campaigns*. All data files in a campaign inherit the campaign's metadata. Access to raw data files is restricted to the owner of the data, unless the owner specifies otherwise.

Raw data is turned into observations by a *normalizer* based on the data format that the raw data is provided in. Normalizers have a simple common interface, allowing them to be implemented in any language or on any platform that can run on UNIX-like systems. A normalizer takes a single raw datafile on standard input, and a metadata stream on file descriptor 3, and produces an *observation file* containing metadata and data in a newline-delimited JSON format on standard output. In addition to normalizing raw data, the normalizer is responsible for protection, producing

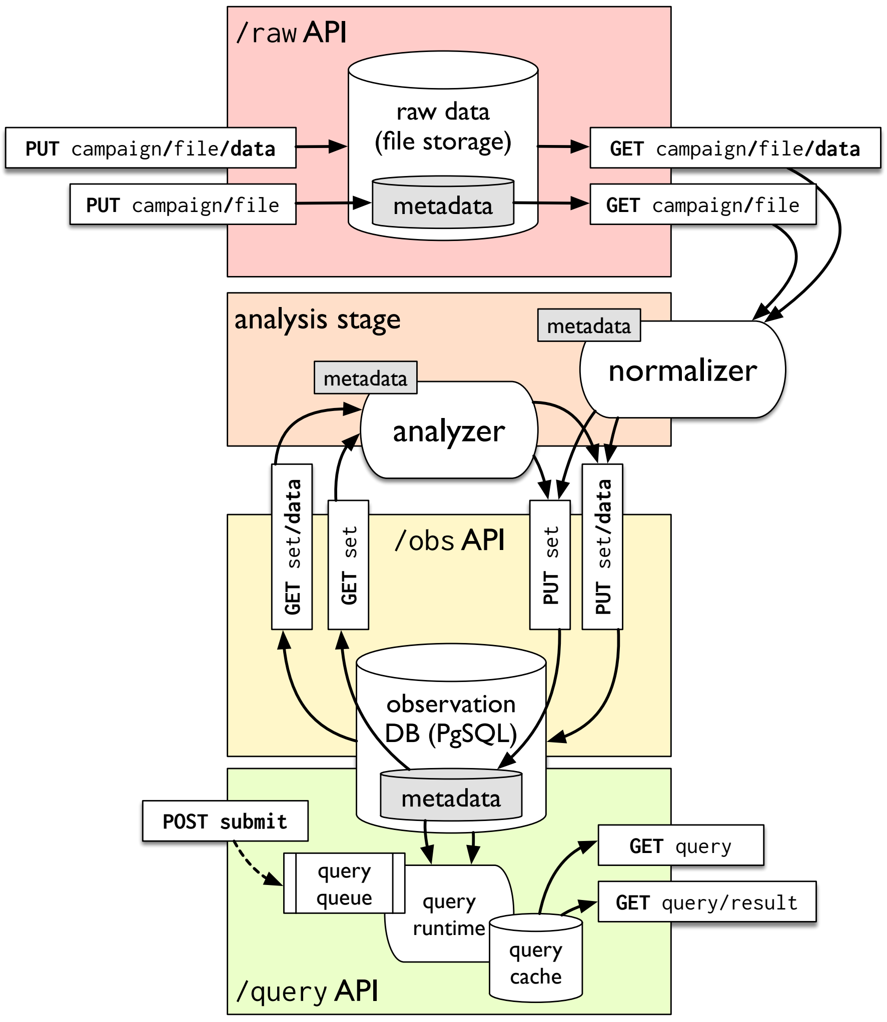

Figure 1: Overview of the PTO Architecture and API

observations that can be disseminated to the entire community of users of a given observatory.

Normalizers are run locally on the observatory, partially automated but with with human intervention. The primary reason for this design choice is review and control: since normalizer code may have been contributed by a data owner distinct from the observatory operator, it is necessary for the operator to have final control over the execution of the normalizers, as well as the insertion of the resulting output into the observation data store.

Observations are grouped into *observation sets*, which share the same source, analyzer, and other metadata. For normalization and analysis purposes, an observation file contains a single observation set. The information model for these observations is task-specific; the one we implemented for path transparency is described in Chapter 3. All data management tasks in the observatory occur at the observation set level, while all observation sets in a given observatory share the same access control. Our implementation of the PTO stores observations in a PostgreSQL database, though the interface is storage technology agnostic.

Observations may also be derived from other observations, e.g., to combine information from multiple vantage points, tools, or campaigns. In this case an *analyzer* retrieves one or more observation sets from the data store, processes them, and produces a derived observation set. As with normalizers, analyzers run with human intervention. Their interface is similarly simple: they take observation files on standard input, and produce an observation file on standard output.

Analyzers and normalizers are described by a metadata file referenced by the observation sets they produce. As this information is statically associated with the analysis executable, it is not generally stored in the observatory, but at a reference maintained by the executable's developer (often in a source control repository or some service associated with it). This metadata describes the file types an analyzer may take as input, the properties of the observations it outputs, and information about the environment it expects to run in.

The main end-user interface to an observatory is through the *query* API. A query is a simple set of predicates evaluated against the whole observation store, with intersection-of-union semantics. Query parameters are task-specific, as they represent properties in the information model. Query results can be in the form of bare observation data, in the case of selection queries, or in the form of groups of aggregates over selected observations.

The observatory supports queries that may take significant computation to produce results: submitting a query places it in a query queue and provides a link to the new query's metadata. When the query is executed, and results are available, this metadata is updated to provide a link to the results. Query metadata and results are cached, so subsequent retrieval or submission of an identical query will use the same information. In addition, queries can be made *permanent*, for example, if they are referenced in a publication, allowing the PTO itself to provide a stable reference for analyzed data.

2.3 Metadata flow

All interactions with the API are metadata-first: to create any object, its metadata must first be created. This design choice was made in the hope that is makes it harder to ignore metadata in data analysis workflows. Object metadata for raw data, observation sets, and queries is represented by the API as JSON objects. The system supports three kinds of metadata:

- *System* metadata contains read-only information about an object generated by the PTO itself, such as its modification time or the size of a data object. System metadata keys begin with “`__`”
- *Reserved* metadata can be written via the API, but it has a specific meaning to the PTO itself, and incorrect or missing values may cause data handling errors. Examples include the filetype of raw data file. Reserved metadata keys begin with “`_`”
- *User* metadata represents arbitrary information about data. It may be used by normalizers and analyzers to control their operaiton. User metadata keys begin with any character other than “`_`”

The base reference to an object in the PTO is the reference to the metadata object, which contains a link to the associated data (when available). Metadata is also used to implement the PTO's provenance functionality. Every observation set's metadata contains links to the raw data and/or observation set(s) from which it was derived (in the `_sources` metadata value), and

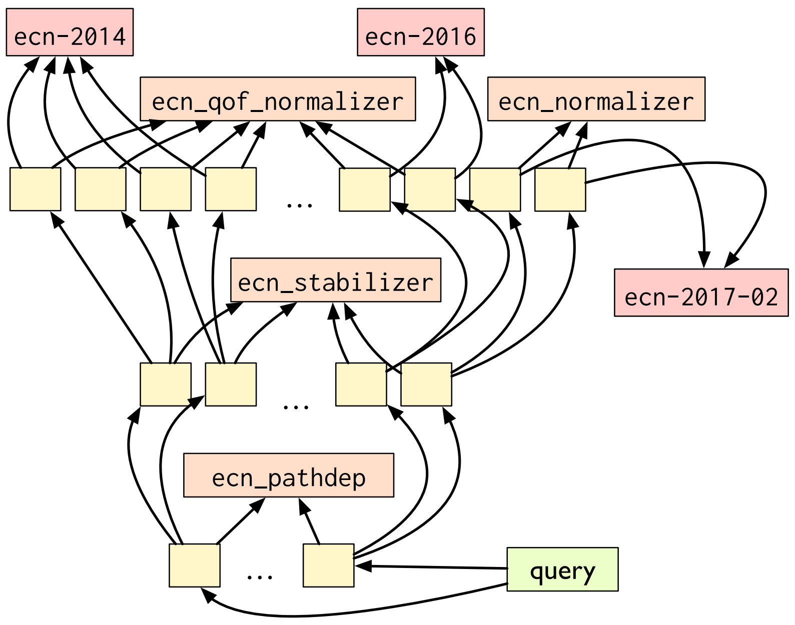

Figure 2: Illustration of PTO provenance graph, made by traversing sources and analyzers. Queries (green) refer to observations (yellow), which in turn refer to less refined observations or raw data (red), and the analyzers or normalizers (orange) that produced them.

metadata describing the analyzer that created it (as `_analyzer`). Query metadata contains links (as `__sources`) to the observation set(s) from which the query's results are derived. By following the links recursively from a query or observation set, the provenance of a given data item can be traced back to the raw source. A provenance graph of the subset of data stored in the PTO is given in Figure 2.

Deprecation is also implemented through metadata; the `_deprecated` metadata value on a raw data file or observation set, if present, contains the time at which an object was marked deprecated. Updating the metadata to contain this key causes the deprecation.

Other system and reserved metadata keys used by the PTO are briefly described in Table 1.

European Commission | Horizon 2020 European Union funding for Research & Innovation

Table 1: System and reserved metadata keys (valid on R = raw data, O = observation set, Q = query)

Key	Valid on	Description
`__created`	ROQ	Creation time
`__modified`	ROQ	Last modification time
`__executed`	Q	Query execution time
`__completed`	Q	Query completion time
`_deprecated`	ROQ	Deprecation time
`_sources`	O(Q)	Source metadata linked
`_analyzer`	O	Analyzer metadata link
`__data`	RO	Link to raw/observation data
`__data_size`	R	Data size in bytes
`__obs_count`	O	Observation count
`_owner`	R	Raw data owner
`_file_type`	R	File type
`_time_start`	R	Start of interval
`_time_end`	R	End of interval
`_conditions`	O	List of conditions present
`__result`	Q	Link to query results
`__state`	Q	State of query
`_ext_ref`	Q	Backlink to permanent reference

European Commission | Horizon 2020 European Union funding for Research & Innovation

3 Observation Information Model

The architecture, data flow, and metadata flow of the PTO are independent of the information model used for observations; however, the utility of the PTO for path transparency measurement is determined by its information model enabling comparability between data from different sources and tools.

Observations within the PTO are stored as four-tuples, as shown in Figure 3, and represent an assertion that a given *condition* was observed during a given *time* interval on a given *path*, and an optional *value* associated with that condition. Observations are grouped into observation sets, by which they hook into the PTO's metadata flow. Observation sets additionally carry a `_conditions` metadata key, which lists the conditions present in each observation set.

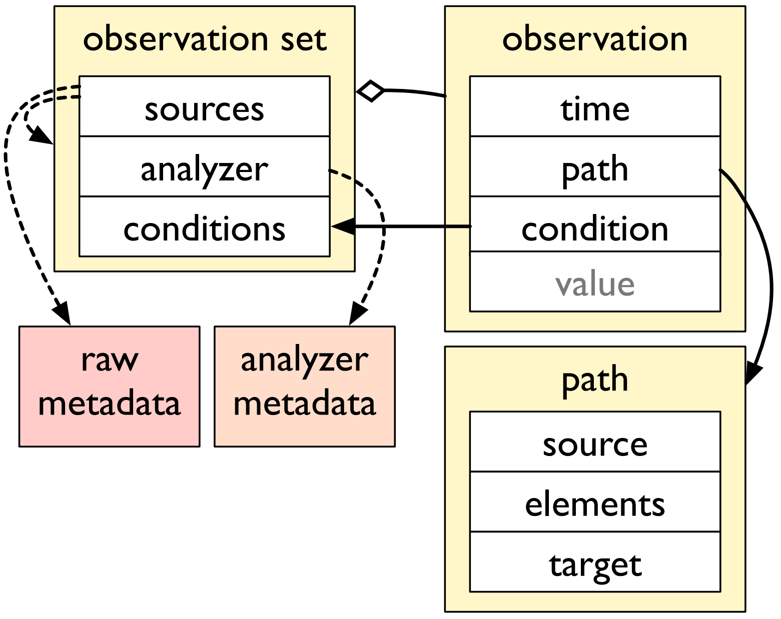

Figure 3: Path transparency information model

3.1 Conditions

A *condition* is something observed, and refers to a specific test derived from active or passive measurement. Condition names are human-readable and structured, beginning with a *feature* being measured, the *aspect* (and zero or more *subaspects*) of that feature, ending with the *state* of that aspect. For example, `ecn.connectivity.works` means the "connectivity" aspect of ECN (i.e., whether attempting to negotiate ECN leads to connectivity failure) has no failure. States on an aspect are mutually exclusive: at any given time for any given observation, only one state holds. The conditions present in the observatory are determined by the normalizers and analyzers that create the observations; normalizers and analyzers declare these conditions in their metadata.

Conditions are treated by the PTO data model as simple strings; only the semantics of the aspect separators are explicitly supported, as queries (see Section 4.3) can select based on condition wildcards. Therefore, the set of conditions appearing in the PTO is determined by the conditions generated by measurement tools and normalizers generating observations in the PTO.

3.2 Paths

A *path* is an ordered sequence of elements; these can be IP addresses, IP prefixes, BGP AS numbers, free-form pseudonyms for path elements or subpaths. This allows the PTO to store data at multiple resolutions; pseudonyms can be used for levels of aggregation other then prefix and AS, and can also grant reasonable protection to internal network topology information, to incentivize otherwise reluctant owners of measurement data to share. Wildcards ($*$) are supported to represent an unspecified number of unspecified hops between two elements.

The following path element formats are currently supported:

Format	Description
the string $*$	zero or more unknown hops on a path
NNN.*NNN*.*NNN*.*NNN*	IPv4 address
NNN.*NNN*.*NNN*.*NNN*/*NN*	IPv4 prefix
[*IPv6 address*]	IPv6 address (see [3])
[*IPv6 address*]/*NN*	IPv6 prefix
AS*NNNNNN*	BGP Autonomous System Number
XXXXX	An arbitrary path element pseudonym

While the use of these path elements is determined by the normalizer or analyser, in our information model we currently have three combinations in use which can be distinguished into the following path types:

Pair path. This kind of path contains a source S and a target T separated by a wildcard: $[S * T]$. This denotes that the condition holds on traffic with the given source and target, but that no other path information is given. Pair paths may be augmented with additional information about the source and target networks; for example, if AS-level information is available (as is the case in some PATHspider runs), this additional information appears between the source/target and the center wildcard: $[S\ AS_S * AS_T\ T]$.

Hop path. This kind of path contains a source S, a target T, and a hop, consisting of the pre- and post-hop path elements A and B. If A and B are in the middle of a long path, then the path will look like $[S * A\ B * T]$. However, several cases may occur for which this general format will look different. The precise rues are given in table 3.

Trace path. This kind of path contains everything that is known about how the packets traveled form the source to the destination. It is simply an ordered sequence of path elements as described above. In particular, several consecutive '$*$' elements may occur, each representing zero or more unknown (i.e., unmeasured) path elements.

Table 3: Rules for forming hop paths. Aassume that the original path form which this hop path was extracted was $[S = P_1\ \ldots\ P_n = T]$, where each P_k is a path element as above. (In particular, any path element may be '$*$'.) We assume $S \neq T$. Let the hop that we are interested in be between nodes k and $k+1$ $(1 \leq k < n)$, and let the final hop path be P. This final hop path is then further compressed by coalescing adjecent '$*$' elements.

If this condition holds...	then P is ...
$n = 2$	$[S\ T]$
$n = 3$ and $k = 1$	$[S\ P_2\ T]$
$n = 3$ and $k = 2$	$[S\ P_2\ T]$
$n > 3$ and $k = 2$	$[S\ P_2 * T]$
$n > 3$ and $k = n - 1$	$[S * P_{n-1}\ T]$
$n = 4$ and $k = 2$	$[S\ P_2\ P_3\ T]$
$n > 4$ and $k = 2$	$[S\ P_2\ P_3 * T]$
$n > 4$ and $k = n - 2$	$[S * P_{n-2}\ P_{n-1}\ T]$
$n > 5$ and $3 \leq k < n - 2$	$[S * P_{k-1}\ P_k * T]$

3.3 Querying the Information Model

The PTO supports queries that look at attributes of the information model. Observations may be selected by time range, presence of an element on the path, or at the beginning (`source`) or end (`target`) of the path, or by the presence of a condition or group of conditions expressed with a wildcard (e.g. `ecn.multipoint.negotiation.*` for all states on the `negotiation` subaspect of ECN observations combining measurements from multiple points). Raw queries return matching observations; queries can also group by predicates on the time interval, sources, targets, and conditions. Query parameters are detailed in table Table 4.

Table 4: Query parameters

Parameter	Type	Mult.	Description
`time_start`	temporal	1	Select observations starting at or after the given start time
`time_end`	temporal	1	Select observations ending at or before the given end time
`set`	select	0..n	Select observations with in the given set ID
`on_path`	select	0..n	Select observations with the given element in the path
`source`	select	0..n	Select observations with the given element at the start of the path
`target`	select	0..n	Select observations with the given element at the end of the path
`condition`	select	0..n	Select observations with the given condition, with wildcards
`group`	group	0..2	Group observations and return counts by *groups*:
`year`			Count by year of `time_start`
`month`			Count by year/month of `time_start`
`day`			Count by year/month/day of `time_start`
`hour`			Count by year/month/day/hour of `time_start`
`week`			Count by year/week (starting Monday) of `time_start`
`week_day`			Count by day of week of `time_start` (7 groups)
`day_hour`			Count by hour of day of `time_start` (24 groups)
`condition`			Count by condition
`source`			Count by first element in path
`target`			Count by last element in path
`option`	options	0..n	Specify a query *option*:
`sets_only`			Return only links to observation sets answering the query
`count_targets`			Count distinct targets in group queries

4 PTO API Definition

Interactions with the PTO are defined by two interfaces: the *REST API* provided by the PTO server `ptosrv`, and the analysis interface provided to normalizers and analyzers for the production of observations. This section documents the API. The analysis interface is described in the next section.

The API consists of three types of resources:

- raw data access and upload (`/raw` paths)
- observation access and upload (`/obs` paths)
- observation database query and query cache (`/query` paths)

The interface to each part of the API is made up of certain resources accessed in a RESTful way; these resources are specified in the subsections below.

This section is taken from the API documentation for our implementation of the PTO at

https://github.com/mami-project/pto3-go/blob/master/doc/API.md

as of this writing. Maintenance of the PTO may involve additions to this API, so check there for the latest documentation.

4.1 Raw Data Access and Upload

The raw data access and upload API (resources under `/raw`) allows the upload of raw data files to the PTO, and the later retrieval of those files. Each file is associated with a *campaign* – a group of files related to a single measurement campaign from a single data source. Campaigns and files have associated *metadata* which is used by the PTO and analysis modules to save metadata about the raw data; this may also be used by users of the PTO to store information about raw files and campaigns.

The resources and methods available thereon are summarized in the table below:

Method	Resource	Description
`GET`	`/raw`	Retrieve URLs for campaigns as JSON
`GET`	`/raw/<c>`	Retrieve metadata for campaign *c* as JSON
`PUT`	`/raw/<c>`	Write metadata for campaign *c* as JSON
`GET`	`/raw/<c>/<f>`	Retrieve metadata for file *f* in *c* as JSON
`PUT`	`/raw/<c>/<f>`	Write metadata for file *f* in *c* as JSON
`GET`	`/raw/<c>/<f>/data`	Retrieve content for file *f* in *c* (by convention)
`PUT`	`/raw/<c>/<f>/data`	Write content for file *f* in *c* (by convention)

Note that the while the system normally provides data storage for a given raw file at a location derived from the file's metadata URL by appending the `/data` path element to that URL, this is not always the case: the actual location of the data is given in the metadata itself. This allows the PTO to refer to external storage for raw data, provided that external storage is URL-addressible and provides the same immutability guarantees as the PTO itself.

4.1.1 Metadata

Associated with each file and each campaign in the raw data store is a *metadata object*. This metadata object is a set of key-value pairs, presented by the API as a JSON object. Certain keys in this object are reserved for use by the system, or are generated by the system. Other metadata keys can be freely used by the creator of a raw data file to communicate metadata information to a future analysis process that will be run on the raw data, or for future retrieval of the file. Metadata key behavior is determined by the metadata key name:

- All metadata keys whose names begin with a single underscore _ are reserved for system use; they may be written to, but have a special meaning to the PTO itself.
- All metadata keys whose names begin with a double underscore __ are virtual, and generated by the system; they may not be written to.
- All other metadata key names are free for use by users and analysis modules.

Files inherit metadata from their containing campaign. If a file's metadata and its containing campaign's metadata have metadata for the same key, the value associated with file overrides that inherited from the campaign for that file.

The following reserved and virtual metadata keys are presently supported:

Key	Description
`_file_type`	PTO filetype. See Filetypes, below.
`_owner`	Identity (via email) of user or organization owning the file/campaign
`_time_start`	Timestamp of first observation in the raw data file, in ISO8601 format
`_time_end`	Time of last observation in the raw data file, in ISO8601 format
`_deprecated`	If present, timestamp at which an observation set was marked deprecated
`__data`	URL of the resource containing file data.
`__data_size`	Size of the file in bytes. 0 if the data file has not been uploaded.

Though the data resource is by convention accessible by appending `/data` to the path of the metadata resource, the system may at any time place data at another path; therefore, clients should only upload data to the path given in the `__data` metadata key.

4.1.1.1 Filetypes

Every raw data file has a *filetype*, given in the `_file_type` key, which the PTO uses to determine how to handle files internally, and which analysis modules use to determine how to read and whether they are interested in raw data files. Each filetype is associated with a MIME type, and the `Content-Type` header on data uploads via PUT must match the filetype associated with the file.

Often, all the files within a campaign will share the same filetype. In this case, filetype information is set in campaign metadata, not in individual file metadata.

While filetypes are extensible, the filetypes supported by the PTO as installed are listed below:

Filetype	MIME type	Description
`obs-bz2`	`application/bzip2`	Compressed observations as observation set files (see Section 5.1)
`obs`	`application/vnd.mami.ndjson`	Uncompressed observations as observation set files

4.1.2 Raw data API usage

We use curl[1] to illustrate the usage of the PTO raw API. We assume the API is rooted at `https://pto.example.com/`; that the API key (see Section 4.4) `abadc0de` holds the permissions `list_raw`, `read_raw:test`, and `write_raw:test`; and that the filetype `test` has been defined in this PTO instance. Note also that in these examples, the output of curl is prettyprinted via `python3 -m json.tool`, not shown.

4.1.2.1 Creating and listing campaigns

To list the campaigns for which raw data is stored, simply fetch the `/raw` resource.

```
$ curl -H "Authorization: APIKEY abadc0de" https://pto.example.com/raw
{
    "campaigns": []
}
```

This PTO instance is empty: no campaigns are stored here. To create the `test` campaign, which we are preauthorized to do, simply upload the campaign's metadata at the campaign's path which, in this example, only contains the mandatory owner and file type tags:

```
$ cat test_campaign.json
{
    "_owner": "you@example.com",
    "_file_type": "test"
}

$ curl -H "Authorization: APIKEY abadc0de" \
       -H "Content-Type: application/json" \
       -X PUT https://pto.example.com/raw/test \
       --data-binary @test_campaign.json
{
    "_file_type":"test",
```

[1]https://curl.haxx.se

```
    "_owner":"you@example.com"
}
```

The reply echoes back the metadata uploaded. A campaign's metadata can be changed by simply uploading new metadata.

We can verify that our campaign has been created by listing campaigns again:

```
$ curl -H "Authorization: APIKEY abadc0de" https://pto.example.com/raw
{
    "campaigns": [
        "http://pto.example.com/raw/test"
    ]
}
```

4.1.2.2 Uploading Raw Data

Once a campaign has been created, uploading raw data to it is a two-step process: creating a new file by uploading its metadata, then uploading the data associated with the file.

For the purposes of this example, we'll upload a single test data file containing some JSON formatted data. First the metadata:

```
$ cat test_metadata.json
{
    "_time_start": "2018-04-25T10:15:35Z",
    "_time_end":   "2018-04-25T10:20:48Z",
    "purpose":     "demonstrate file upload"
}

$ curl -H "Authorization: APIKEY abadc0de" \
       -H "Content-Type: application/json" \
       -X PUT https://pto.example.com/raw/test/test001.json \
       --data-binary @test_metadata.json
{
    "__data": "https://pto.example.com/raw/test/test001.json/data",
    "_file_type": "test",
    "_owner": "you@example.com",
    "_time_end": "2018-04-25T10:20:48Z",
    "_time_start": "2018-04-25T10:15:35Z",
    "purpose": "demonstrate file upload"
}
```

Here the uploaded metadata, including keys inherited from the campaign, is echoed back from the server, along with a link to which data can be uploaded (in the `__data` key).

Now that the metadata is created, we can upload the data file to the given URL:

```
$ curl -H "Authorization: APIKEY abadc0de" \
       -H "Content-Type: application/json" \
       -X PUT https://pto.example.com/raw/test/test001.json/data \
        --data-binary @test_data.json
{
    "__data": "https://pto.example.com/raw/test/test001.json/data",
    "__data_size": 37,
    "_file_type": "test",
    "_owner": "you@example.com",
    "_time_end": "2018-04-25T10:20:48Z",
    "_time_start": "2018-04-25T10:15:35Z",
    "purpose": "demonstrate file upload"
}
```

This echoes back the metadata for the file. Note here the new `__data_size` key, which gives the size of the data file in bytes.

4.1.2.3 Downloading Raw Data

While the current PTO implementation by convention always generates data URLs from metadata URLs by appending `/data` to the path, this is not guaranteed to always be the case, so it's important to check the `__data` key in the metadata for the file before downloading. Here we assign this to a shell variable, then download from that url:

```
$ DATAURL=`curl -s -H "Authorization: APIKEY abadc0de" \
                https://pto.example.com/raw/test/test001.json | \
          python3 -c 'import sys, json; print(json.load(sys.stdin)["__data"])'`
$ curl -H "Authorization: APIKEY abadc0de" $DATAURL > downloaded_file.json
```

4.1.2.4 Changing Metadata and Data

Metadata can be changed by uploading a new metadata object.

Once a file has been uploaded, its data can no longer be changed.

Files can currently not be deleted via the API, though a file can be fully deleted by removing the file and metadata file from the filesystem backing the raw data store. To mark a file (or a campaign) as no longer valid, the `_deprecated` system metadata tag is used.

4.2 Observation

The observation access API (resources under `/obs`) allows access to PTO *observations*, whose information model is described in Chapter 3.

Observations are grouped into *observation sets*. An observation set is a set of observations resulting from a single run of an analyser on some input data (see Data Analysis, below). All observations in an observation set share the same metadata and *provenance*. Provenance provides information about the source data that was analyzed (in terms of raw data files and/or other observation sets stored in the PTO) and the analysis that was performed.

European Commission | Horizon 2020 European Union funding for Research & Innovation

The resources and methods available thereon are summarized in the table below:

Method	Resource	Description
`GET`	`/obs`	Retrieve URLs for observation sets as JSON
`GET`	`/obs/by_metadata`	Retrieve URLs for observation sets by metadata
`GET`	`/obs/conditions`	List conditions in observation database
`POST`	`/obs/create`	Create new observation set
`GET`	`/obs/<o>`	Retrieve metadata and provenance for *o* as JSON
`PUT`	`/obs/<o>`	Update metadata and provenance for *o* as JSON
`GET`	`/obs/<o>/data`	Retrieve obset file for *o* as NDJSON (by convention)
`PUT`	`/obs/<o>/data`	Upload obset file for *o* as NDJSON (by convention)

4.2.1 Metadata and Provenance

As with raw data files, observation sets have associated metadata; as with raw data files, arbitrary metadata can be set on observation sets by the analyses that create them. The same rules for reserved and virtual metadata names apply for observation sets as for raw data. The metadata for an observation set also contains information about the observation set's provenance.

Provenance is tracked by three metadata keys. `_sources` is an array of URLs referencing the raw data file or the observation set(s) from which the observations in an observation set are derived. `_analyzer` is a URL referring to the analyzer that created the observation set. `_campaign` is a URL referring to the campaign from which the original raw data was derived, and is only present if the observation set is derived only from observation sets / raw data in the same campaign.

The following reserved and virtual metadata keys are presently supported:

Key	Description
`_sources`	Array of PTO URLs of raw data sources and observation sets
`_analyzer`	URL of analyzer metadata
`_conditions`	Array of conditions declared in the observation set
`_deprecated`	If present, timestamp at which an observation set was marked deprecated
`__obs_count`	Count of observations in the observation set
`__data`	URL of the resource containing observation set data

European Commission | Horizon 2020 European Union funding for Research & Innovation

4.2.2 Querying Observation Sets by Metadata

The `/obs/by_metadata` resource lists links to Observation Sets based on the presence or value of metadata keys, or on the values of particular metadata. The following query parameters are supported:

Key	Description
`k`	Obsets containing metadata key (with `v`, of a specific value)
`v`	Value to query (use with `k`)
`source`	Obsets derived from a source URL starting with a given prefix
`analyzer`	Obsets derived from an analyzer whose metadata URL starts with a given prefix
`condition`	Obsets declaring a given condition

When multiple parameters are given, the intersection of observation sets fulfilling all parameters is returned.

4.3 Query API

The observation query API (resources under `/query`) allows the submission of *queries* to the PTO, to retrieve observations and observation aggregates meeting certain criteria from the PTO's observation database. The resources and methods thereon are summarized in the table below:

Method	Resource	Description
`POST` or `GET`	`/query/submit`	Submit a query
`GET`	`/query`	List currently cached and pending queries
`GET`	`/query/<q>`	Get query metadata, including ETA for pending queries
`GET`	`/query/<q>/result`	Get query results (by convention)
`PUT`	`/query/<q>`	Update query metadata

Queries can be submitted by POSTing to the /query/submit resource. The query itself is defined by a the parameters in the POSTed `application/x-www-form-urlencoded` content. The parameters are summarized below:

Parameter	Kind	Multiple?	Meaning
`time_start`	temporal	no	Select observations starting at or after the given start time
`time_end`	temporal	no	Select observations ending at or before the given end time
`set`	select	yes	Select observations with in the given set ID
`on_path`	select	yes	Select observations with the given element in the path
`source`	select	yes	Select observations with the given element at the start of the path
`target`	select	yes	Select observations with the given element at the end of the path
`condition`	select	yes	Select observations with the given condition, with wildcards
`group`	group	yes	Group observations and return counts by group
`option`	options	yes	Specify a query option

All parameters with temporal semantics must be present, and are used to bound the query in time. Parameters with select semantics may be given to filter observations. if multiple instances of a select parameter are available, any of the values will match; however, an observation must match at least one of the values for each distinct parameter given (i.e., the query language supports AND of OR semantics). Parameters with group or set semantics, as well as the option parameter, may modify the type of query and the format of its results; see the Results section below.

4.3.1 Query Options

The `option` parameter is used to modify the behavior of queries. Multiple Options may be present. The following options are presently supported:

Option Value	Behavior
`sets_only`	Return links to observation sets containing observations answering the query, instead of observation data directly
`count_targets`	Group queries should count distinct targets, not distinct observations

4.3.2 Metadata

When a query is submitted, it goes into the query cache. The query cache holds the query metadata until the query has been scheduled to run. Once it has run, the query metadata will updated to point to the result and the observation sets the query answers.

Key	Description
`__encoded`	URL-encoded parameters from which the query was generated
`__state`	Query state; see below
`__link`	URL pointing to canonical query metadata, when available
`__result`	URL of the resource containing complete result, when available
`__sources`	Array of PTO URLs of observation sets covered by the query, when available
`_ext_ref`	External reference for a permanence request; see below

A query can have one of following states:

State	Meaning
`submitted`	Submitted, but not yet running
`pending`	Running and awaiting results
`failed`	Abnormally ended without returning results
`complete`	Results are available
`permanent`	Results are available and cached results will be stored permanently

4.3.3 Results

The type of the query determines the format of the results, as below:

4.3.3.1 Observation Selection Queries

A query created without any `group` parameters and without the `sets_only` option is a selection query. The observations returned by this query are those within the interval between the `time_start` and `time_end` parameters which match the selection parameters.

The result of a selection query is a JSON object, the fields of which are as follows:

Key	Value
`prev`	Link to previous page (see Pagination)
`next`	Link to next page (see Pagination)
`obs`	JSON array containing observations in OSF format (see Section 5.1)

4.3.3.2 Observation Set Selection Queries

A query created without any `group` parameters and with the `sets_only` option is a selection query. The observations returned by this query are those within the interval between the `time_start` and `time_end` parameters which match the selection parameters.

Key	Value
`prev`	Link to previous page (see Pagination)
`next`	Link to next page (see Pagination)
`sets`	JSON array containing links to observation sets containing observations answering the query

4.3.3.3 Aggregation Queries

A query created with one or more `group` parameters is an aggregation query. The results will return the count of obsevations selected by the select parameters for each group parameter. The following `group` parameters are available:

Value	Meaning
`year`	Count by year of time_start
`month`	Count by year/month of time_start
`day`	Count by year/month/day of time_start
`hour`	Count by year/month/day/hour of time_start
`week`	Count by year/week (starting Monday) of time_start
`week_day`	Count by day of week of time_start (7 groups)
`day_hour`	Count by hour of day of time_start (24 groups)
`condition`	Count by condition
`feature`	Count by feature (first component of condition)
`value`	Count by condition value
`source`	Count by first element in path
`target`	Count by last element in path

The result of an aggregation query is a JSON object, the fields of which are as follows:

Key	Value
`prev`	Link to previous page (see Pagination)
`next`	Link to next page (see Pagination)
`groups`	List of JSON arrays containing count in final position, by group(s)

4.4 Access Control and Permissions

All applications use API key based access control. An API key is associated with a set of *permissions*, scoped to permitted operations on the observatory database, on access to raw data for specific campaigns. The following permissions are defined:

Permission	Description
`list_raw`	List campaign URLs
`read_raw:<c>`	Read raw data and metadata for campaign *c*
`write_raw:<c>`	Write raw data and metadata for campaign *c*
`read_obs`	List observations, read observation data and metadata
`write_obs`	Write observation data and metadata
`submit_query`	Submit all kinds of queries
`submit_query_group`	Submit queries with a `group` parameter
`submit_query_obs`	Submit queries without a `group` parameter
`read_query`	Read query data and metadata
`update_query`	Update query metadata

Raw data access is scoped to campaign; in the table above, `<c>` is the name of the campaign to which the condition pertains. Observation and query access is scoped to the entire instance.

API keys are given in the HTTP `Authorization` request header, which must consist of the string `APIKEY` followed by whitespace and the API key as a string.

If no API key is given, a request is evaluated against a configured default set of permissions. In this way, a given instance of the PTO can support both public and access-controlled usage.

Horizon 2020
European Union funding
for Research & Innovation

5 PTO Analysis Interface Definition

This chapter describes the current state of the PTO analysis interface at the time of writing; it is derived from documentation available at

https://github.com/mami-project/pto3-go.

The PTO stores raw data, but allows queries over observations in a database. These observations are created by *normalizers* and *analyzers*. The former take raw data as input, the latter take observation sets as input. The interface to analyzers is designed to be as implementation-independent as possible.

Normalizers take in raw data and output observation sets in the file format described in Section 5.1. Raw data is passed in on standard input, and observations and metadata on standard output. Since raw data and metadata cannot be interleaved in the general case, these are passed in as a JSON file on file descriptor 3. A single invocation over a raw data file will result in a single observation set.

Analyzers simply take in observation data and metadata on standard input in observation set format, and output observation data and metadata in the same format on standard output. Observations from multiple observation sets may appear in the input to an analyzer.

Normalizers and analyzers are described by metadata whose format is defined in Section 5.2.

5.1 Observation Set File Format

Observation sets are accessible in the PTO API, and provided to and generated by analyzers in the analyzer interface, as Observation Set Files (OSF). An OSF is a newline-delimited JSON (ndjson) file where each line contains either an observation or an observation set metadata objects.

5.1.1 Data Elements

JSON arrays in the file are treated as observations. An array has five or six elements, with the following semantics and format:

Position	Description
0	Observation Set Identifier, a string
1	Start time in RFC 3339 format
2	End time in RFC 3339 format
3	Path, as defined below
4	Condition, as a JSON string
5	Value associated with condition, as a JSON string; optional

A *path* is a sequence of path elements. It can be represented either as a JSON array of strings, each one a path element; or as a JSON string containing a whitespace-separated sequence of path elements. The PTO currently generates only space-separated path strings, but accepts and may generate either in the future.

Path elements identify hops along the path. The type of path element is implied by its format, as described in Section 3.2.

A *condition* is fundamentally a free-form string; however, the convention presently used in the PTO uses a hierarchical structure for condition names. A condition name is made up of condition name elements separated by `.` characters, where the first element describes the protocol, extension, feature, or metric to which the condition relates, intermediate levels refer to aspects or codepoints of the tested feature, and the final level refer to the condition itself. For example, `ecn.negotiation.succeeded` means that ECN negotiation was attempted and appeared to be successful.

5.1.2 Metadata Elements

JSON objects in the file are treated as metadata key-value pairs, depending on context.

Any ndjson file containing 5- or 6-element arrays that can be interpreted as observations and objects that can be interpreted as metadata is considered a well-formed obsetvation set file. However, in various contexts, the PTO provides additional contracts on the file format, as below:

5.1.2.1 Observation Access API

With Observation Access API, the observation set ID is filled in on download, and ignored on upload. Metadata is not present in downloaded files, and is ignored in uploaded files.

5.1.2.2 Results via Query API

Within a query result, the `obs` key contains a JSON array of observations as in this file format. The observation set ID is present and refers to the observation set ID within the PTO. Observation set metadata is not present.

5.1.2.3 As Analyzer Input

When provided to a local analyzer as input, observation set IDs are significant: all observations with the same observation set ID are taken to be members of the same observation set. Metadata is present, and appears directly before the first observation in each observation set.

5.1.2.4 As Analyzer Output

When produced by a local analyzer as output, observation set IDs are ignored. Multiple metadata elements may be present, but only the last metadata element present will be taken as metadata for the new observation set.

European Commission | Horizon 2020 European Union funding for Research & Innovation

5.2 Analyzer Metadata

Analyzer metadata, referred to in an observation set's `_analyzer` metadata key, is represented as a JSON object, which has the following keys:

Key	Description
`_repository`	URL of source code repository, if not implicit
`_owner`	Identity (via email) of user or organization owning the analyzer
`_file_types`	File types consumable by raw analyzer, as array
`_invocation`	Command to run in repository root to invoke the analyzer, if local
`_platform`	Platform identifier (one of ‘python‘, ‘go‘, or ‘shell‘ with an optional version identifier), implying how to set up the analyzer environment

As with raw and observation metadata, all keys not beginning with _ are freeform, and may be used to store other information about the analyzer.

Normalizer metadata must contain a `_file_types` key, a JSON array listing PTO filetypes it can consume; analyzer metadata must not contain this key.

Analyzers can take one of two forms. A *local* analyzer is designed to run within the PTO, with direct access to raw data files or observations. It reads raw data or observation files on standard input, and produces observation files and observation set metadata on standard output. Local analyzer metadata must contain an `_invocation` key, which is a command to run in the repository root to invoke the analyzer. Local analyzers can be run manually on a machine with access to a PTO instance's raw data and observation database using the command-line tools described in Section 6.3.

A *client* analyzer is designed to use the PTO API to retrieve raw data and observation sets and upload its results. As it cannot be automatically invoked, client analyzer repositories should contain human-readable documentation for invocation. Client analyzer metadata must not contain an `_invocation` key.

6 PTO Implementation and Deployment

This section describes the current state of our PTO implementation and the normalizers and analyzers we use at the time of writing; it is derived from documentation available at

https://github.com/mami-project/pto3-go.

6.1 PTO API Server

We run an instance of the PTO at

https://v3.pto.mami-project.eu,

implemented in Go, using flat files as backend storage for raw data and cached queries, and a PostgreSQL database for storing and querying observations. The codebase and documentation is available at

https://github.com/mami-project/pto3-go.

6.2 PTO Analysis Runtime

To populate our observation database from raw data, we use the PTO analysis runtime provided with our PTO server codebase. The PTO comes with a set of command-line tools for running normalizers and analyzers locally (i.e., on the same machine running `ptosrv`, or on a machine with equivalent access to the raw filesystem and the PostgreSQL database).

Three tools are provided:

- `ptonorm`: read data and metadata from raw data store, hadling campaign metadata inheritance, run a normalizer, and pipe to stdin / fd 3.
- `ptocat`: dump observation sets from the database with metadata to stdout
- `ptoload`: read files with observation set data and metadata and insert resulting observation sets into database

6.2.1 Running Normalizers

Local normalizers are run by `ptonorm`, which takes the following command-line arguments:

```
ptonorm -config <path/to/config.json> <normalizer> <campaign> <file>
```

If `-config` is not given, the file `ptoconfig.json` in the current working directory is used.

`ptonorm` launches the normalizer as a subprocess, allowing access to the raw data file over stdin, and streaming metadata over a pipe on file descriptor 3. It then takes the standard output, coalescing all metadata into a single object, and writes it to standard output. When coalescing metadata, the last write on a given metadata key wins.

The resulting observation file can be passed as input to `ptoload`, which takes the following command-line arguments:

```
ptoload -config <path/to/config.json> <obsfile>...
```

If `-config` is not given, the file `ptoconfig.json` in the current working directory is used. More than one observation file can be given on a single command line, but each file given will create a new observation set.

For example, to normalize the file `quux.ndjson` with the `bar` normalizer in the `foo` campaign into an observation set, using a local configuration file, and load it directly into the database, deleting the cached observation file:

```
ptonorm bar foo quux.json > cached.obs && ptoload cached.obs && rm cached.obs
```

6.2.2 Running Analyzers

Analyzers are simpler to run, as they take observation files on standard input and generate observation files on standard output. To get observation files, use `ptocat`, which takes the following command line arguments:

```
ptocat -config <path/to/config.json> <set-id>...
```

If `-config` is not given, the file `ptoconfig.json` in the current working directory is used. Set IDs are given in hexadecimal, as in the rest of the PTO. More than one set ID may appear; in this case, the metadata for the first set will be followed by the data for the first set followed by the metadata for the second set followed by the data for the second set and so on.

For example, to analyze sets 3a70 through 3a72 using the analyer `fizz` and load it directly into the database, deleting the cached observation file:

```
ptocat 3a70 3a71 3a72 | fizz > cached.obs && ptoload cached.obs && rm cached.obs
```

6.3 PTO Normalizers/Analyzers and Conditions

We maintain two repositories with normalizers and analyzers for our own use.

Tools for normalizing and analyzing PATHspider output (originally focused on ECN, but supporting all plugins as of pathspider version 2.0) are in the following repository:

https://github/mami-project/pto3-ecn

Tools for dealing with Tracebox output, which can be generalized for other traceroute-similar input data formats, are in this repository:

https://github/mami-project/pto3-trace

Conditions present in our instance of the PTO, i.e., those generated by tools maintained by the project, are determined by the normalizers and analyzers we run. They are detailed in Table 23, Table 24, Table 25, and Table 26.

Table 23: Hierarchy of conditions supported by the PTO for ECN

Condition	Value	Description
`ecn`		Explicit Congestion Notification tests
`.connectivity`		Connectivity dependency tests
`.works`	-	Connectivity not dependent on attempt to negotiate ECN
`.broken`	-	Attempt to negotiate ECN leads to connectivity failure
`.offline`	-	Target offline regardless of ECN connectivity
`.transient`	-	Connectivity only with ECN negotiation; probable transient
`.negotiation`		
`.succeeded`	-	ECN negotiated successfully
`.failed`	-	ECN not negotiated
`.reflected`	-	ECN negotiation reflects ECE CWR flags on SYN ACK
`.stable`		
`.connectivity`		Stabilized connectivity dependency tests
`.works`	obs. count	`ecn.connectivity.works` on multiple observations
`.broken`	obs. count	`ecn.connectivity.broken` on multiple observations
`.offline`	obs. count	`ecn.connectivity.offline` on multiple observations
`.transient`	obs. count	`ecn.connectivity.transient` on multiple observations
`.unstable`	-	No consensus about connectivity across multiple observations
`.negotiation`		Stabilized negotiation tests
`.succeeded`	obs. count	`ecn.negotiation.succeeded` on multiple observations
`.failed`	obs. count	`ecn.negotiation.failed` on multiple observations
`.reflected`	obs. count	`ecn.negotiation.reflected` on multiple observations
`.unstable`	-	No consensus about negotiation across multiple observations
`.multipoint`		
`.connectivity`		Multiple vantage point connectivity dependency tests
`.works`	VP count	`ecn.connectivity.works` from all vantage points
`.broken`	VP count	`ecn.connectivity.broken` from all vantage points
`.offline`	VP count	`ecn.connectivity.offline` from all vantage points
`.transient`	VP count	`ecn.connectivity.transient` from all vantage points
`.path_dependent`	VP count	Evidence that connectivity dependency is path dependent
`.negotiation`		Multiple vantage point negotiation tests
`.succeeded`	VP count	`ecn.negotiation.succeeded` from all vantage points
`.failed`	VP count	`ecn.negotiation.failed` from all vantage points
`.reflected`	VP count	`ecn.negotiation.reflected` from all vantage points
`.path_dependent`	VP count	Evidence that negotiation is path dependent
`.ipmark`		IP codepoint on flow from server
.codepoint		one of `ce`, `ect0`, `ect1`
`.seen`	-	given codepoint was observed
`.not_seen`	-	given codepoint was not observed

Table 24: Hierarchy of conditions supported by the PTO for other Pathspider tests

Condition	Value	Description
`dscp`		Differentiated Services tests (single point)
.codepoint		Tests comparing *codepoint* (decimal 0-63) on SYN to DSCP 0
`.connectivity`		DSCP connectivity dependency tests
`.works`	-	Connectivity not dependent on given DSCP on SYN
`.broken`	-	Setting given DSCP on SYN causes connectivity failure
`.offline`	-	Target offline regardless of DSCP on SYN
`.transient`	-	Connectivity only with given DSCP on SYN, possible transient
`.replymark`	DSCP	DSCP mark (decimal 0-63) seen on SYN ACK
`h2`		HTTP/2 tests
`.connectivity`		H2 connectivity dependency tests
`.works`	-	Both H1 and H2 supported
`.broken`	-	Attempts to negotiate H2 lead to connectivity failure
`.offline`	-	Target offline regardless of HTTP version
`.transient`	-	Only H2 supported, possible transient
`.upgrade`		H2 upgrade test
`.success`	-	Upgrade to H2 successful
`.failed`	-	Upgrade to H2 failed
`tfo`		TCP Fast Open tests
`.connectivity`		TFO connectivity dependency tests
`.works`	-	Connectivity not dependent on TFO
`.broken`	-	Attempts to negotiate TFO lead to connectivity failure
`.offline`	-	Target offline regardless of TFO option
`.transient`	-	Connectivity requires TFO option, probable transient
`.cookie`		TFO cookie conditions
`.received`		TFO cookie received
`.not_received`		TFO cookie not received
`.syndata`		Tests for data on SYN with TFO cookie
`.acked`		TFO data on SYN ACKed
`.not_acked`		TFO data on SYN not ACKed
`.failed`		TFO data on SYN causes RST or no response
`udpzero`		DNS with UDP zero checksum connectivity tests
`.connectivity`		
`.works`	-	Zero checksum packets receive a reply
`.broken`	-	Zero checksum packets receive no reply
`.offline`	-	Target offline regardless of UDP checksum
`.transient`	-	Only zero checksum packet receives reply, probable transient

European Commission | Horizon 2020 European Union funding for Research & Innovation

Table 25: Hierarchy of conditions supported by the PTO for Tracebox data, part 1. Any `.changed` condition contains in its value the new value of the field. For example, if the condition is `tcp.option.mss.changed`, the condition's value is the new MSS value. For a list of TCP options, see [13]

Condition	Description
`dscp`	Explicit Congestion Notification tests
.codepoint	Tests receiving *codepoint* (decimal 0-63) on SYN
`.changed`	DSCP mark (decimal 0–63) seen on ICMP message
`tcp`	Tests involving TCP
`.option.`*opt*`.changed`	TCP option value changed in ICMP time-exceeded message
`.ws`	Window size option
`.ts`	Timestamp option
`.mss`	Maximum segment size option
`.sack`	Selective ACK request option
`.sackok`	Slective ACK option
`.ao`	Authentication option
`.rfc1072.echo`	Long-Delay Path option
`.rfc1644.cc`	T/TCP options
`.rfc1644.echo`	T/TCP options
`.rfc1644.new`	T/TCP options
`.md5`	MD5 Authentication option
`.rfc4782`	Quick-Start Response option
`.rfc1072.reply`	TCP Echo Reply option
`.rfc1693.permitted`	TCP Partial Order Service option
`.rfc1146.request`	TCP Alternate Checksum options
`.snap`	TCP SNAP option
`.user-timeout`	TCP user timeout option
`.trailer-checksum`	TCP trailer checksum option
`.scps-capabilities`	SCPS capabilities option
`.rfc1146.data`	Alternate checksum option
`.rfc1693.profile`	TCP partial order service profile option
`.selective-nack`	Selective negative acknowledgments
`.record-boundaries`	TCP record boundaries
`.mptcp`	Multipath TCP option
`.corruption-experienced`	Corruption experienced option

European Commission | Horizon 2020 European Union funding for Research & Innovation

Table 26: Hierarchy of conditions supported by the PTO for Tracebox data, part 2.

Condition	Description
`ecn`	Explicit Congestion Notification options
`.ip.changed`	ECN IP signaling seen
`ip`	Changes seen in the IP header
`.flags.changed`	IP flags have changed
`ip4`	Changes seen in the IP version 4 header
`.id.changed`	IPv4 ID field changed
`tcp.`*field*`.changed`	TCP header field changed
`ack`	Acknowledgment number
`flags`	Flags
`length`	Length
`offset`	Offset
`reserved`	Reserved bits
`sport`	Source port
`urg`	Urgent pointer
`window`	Window size

6.4 Path Transparency Measurement Tools

Raw data for the conditions in Table 23 and Table 24 is generated by PATHspider, and raw data for the conditions in Table 25 and Table 26 is generated by tracebox. Both of these tools were detailed in Deliverable 1.1.

In the meantime, PATHspider has evolved slightly. First, its output has been changed to be more easily normalizable, generating timestamps, paths, and conditions trivially transformable into PTO observations, for better integration with the PTO. Second, its design has become more generalized based on our experience using it for measurement. We discuss the current state of this design in the subsections below, with reference to the architecture diagram in Figure 4, as of PATHspider's 2.0 release.

The main difference for the architecture is that, for the 2.0 series, PATHspider was extended to allow for arbitrary numbers of tests to be performed as opposed to a single A/B test. This was done through the addition of an extra *Combiner* stage before the output stage that would collect the results from individual tests and then only combine them with the job records to produce path conditions once all the tests had completed for a target. Job records are passed to the combiner stage independent of the results of the workers to reduce memory consumption, as each job record can contain metadata that can assist during analysis but should not be duplicated for each test connection. In this new architecture, the API available to plugins becomes more flexible.

Additional new features and extensions that are supported by this architecture are described in the next sections.

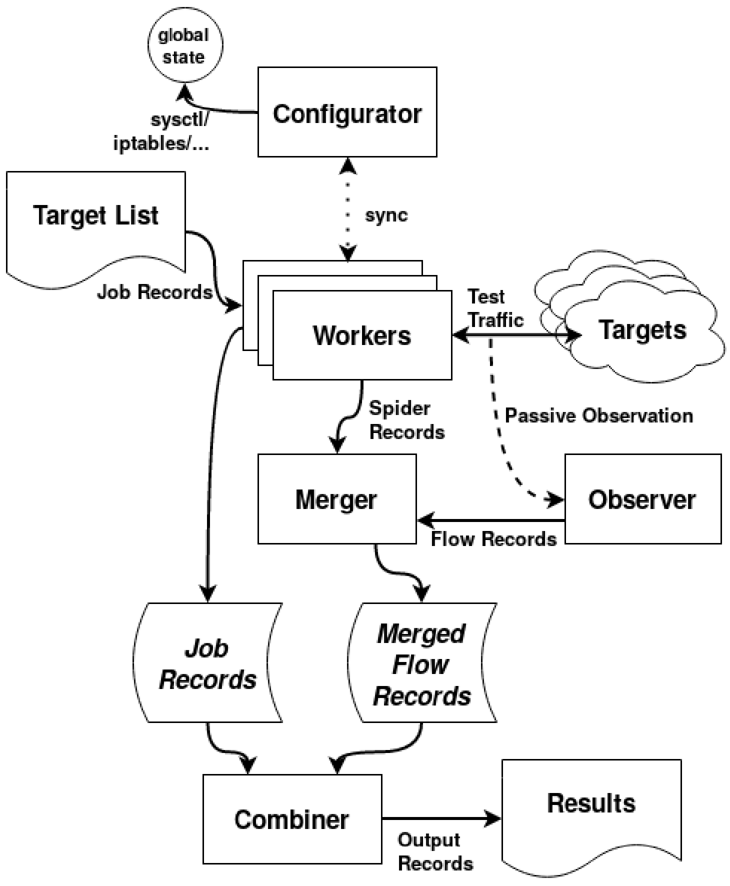

Figure 4: An overview of the PATHspider 2.0 architecture

6.4.1 Auxiliary Network Information using Hellfire

PATHspider now has a companion tool, called Hellfire, to perform high-speed DNS resolution to turn one of a number of public target lists into resolved IP address lists, easing the generation of input for large-scale PATHspider runs, available here:

https://github.com/mami-project/hellfire

In addition, PATHspider integrates with the CANID tool, also developed within the MAMI project, to associate target lists with additional information about the targeted IP addresses. This is used to add BGP autonomous system number information to PATHspider-generated paths in the PTO. See here:

https://github.com/britram/canid

6.4.2 PATHspider Extensibility

PATHspider's extensibility framework has been streamlined for the 2.0 release in January 2018. Each kind of path impairment, that PATHspider can measure, is handled by a plugin dedicated to that type of measurement. Plugins can be one of three types: synchronized, desynchronized, or forging; the type of a plugin determines how traffic is generated.

Synchronized plugins are used for impairments that require system-level changes to be made to configuration between the control and experimental connections being made, for example ECN testing, which uses `sysctl` to change the kernel's ECN negotiation behavior between tests: these are synchronized as only either control or experimental connections can be made at any given time. Desynchronized plugins do not have this constraint, and therefore require no system-level locking among different traffic types. Thus if not require by the measurement, this desynchronization significantly speeds up PATHspider.

Further, forging plugins do not use system APIs at all to generate traffic, instead they support easy forging of customized packets using Scapy. Traffic generation runs in the worker threads, and plugins there can track their state in the job record passed as input from the target queue.

Regardless of type, each plugin defines a set of observer chain functions, which run in the observer on a per-packet basis, and know how to extract data relevant to each measurement from each flow observed. Plugins can implement arbitrary functions for the observer function chain, or reuse library functions for some functionality. 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 for IP, TCP and UDP packets allow different behaviours based on the IP version and transport protocol.

Each plugin must also provide a merge function and a combiner function. The former takes data from a final job record and merges it with the relevant flow record, and the latter combines flows for multiple states in order to generate one or more PTO conditions for the measurement taken.

6.4.3 PATHspider-tracebox Integration

In order to examine PATHspider results in the context of the actual paths traversed by upstream traffic, as well as to attempt to correlate visible on-path impairments with end-to-end impairments, we integrated route tracing and impairment localization into PATHspider based upon the tracebox [6] methodology. Through the integration with PATHspider we can use information derived from A/B testing to find paths on which a given feature does not work, and immediately trace the path on which the feature failed. In contrast to a two-phase approach taking the results from PATHspider and feeding them into a separate Tracebox executable, this approach has less delay between the initial detection and the follow-up traceroute measurement, and can easily use a TCP packet with the same characteristics of that which caused the measured failure.

To realize this change, we created two new modules: a traceroute sender and a traceroute merger, as shown in Figure 5. When a measurement for a target has returned a result that warrants further investigation, a traceroute for this target is performed by passing the target details to the traceroute sender. This will use a forged packet specific to the kind of measurement to attempt to make broken behaviour observable. For example, for ECN it may be the case that a middlebox on the path clears the ECN-Capable Transport (ECT) field in the IP header and so the traceroute would be performed with this field set to “all ones”, Congestion Experienced (CE), and it will be observable when the field is cleared. If a flow contains ICMP Time-To-Live (TTL) exceeded packets, it is marked for analysis by the traceroute merger module. This module will prepare the traceroute data for output.

European Commission
Horizon 2020
European Union funding
for Research & Innovation

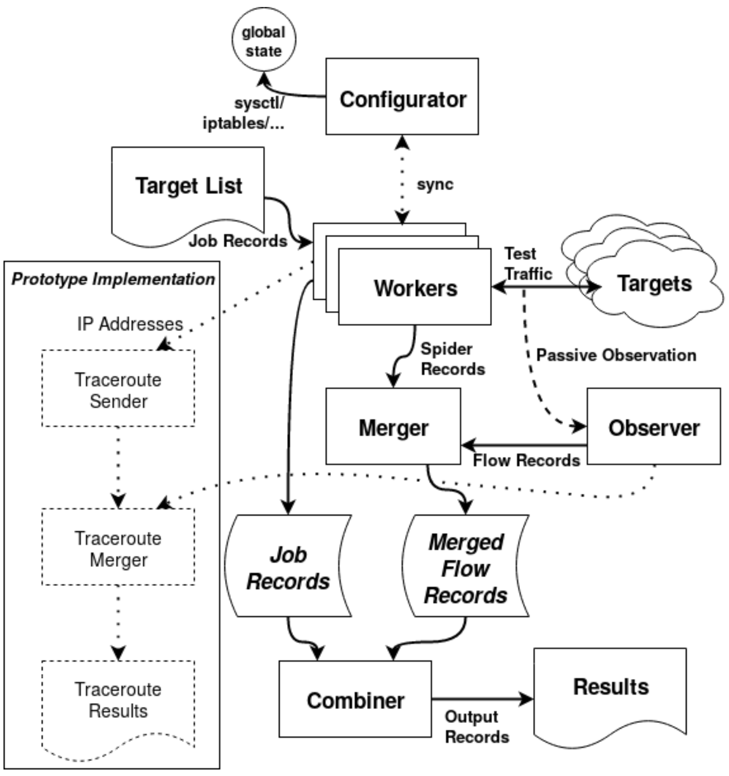

Figure 5: Extended architecture of PATHspider 2.0 including Tracebox prototype modules.

This prototype implementation was used to run a measurement study published and presented at the TMA conference [12]; however, integration of this functionality into the master branch of PATHspider requires at this point some refactoring, as the development of the prototype has been started before the 2.0 release. With the 2.0 architecture, traceroutes can now be implemented as additional connections and the merging of traceroute results can be handled by the combiner. Existing plugins for passive observation can be easily extended to act on the ICMP quotations without the need for this to happen in the merging module. This refactoring is still on-going.

6.4.4 PATHspider Plugins

A number of plugins have been developed for PATHspider by MAMI. All of the following plugins ship with PATHspider 2.0 by default.

6.4.4.1 ECN Plugin

The ECN plugin for PATHspider aims to detect breakage in the Internet due to the use of Explicit Congestion Notification (ECN) [18]. ECN is an extension to the Internet Protocol and to the Transmission Control Protocol. 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% [21]. The ECN PATHspider plugin supports measurement ECN-based connectivity failures as well as observing the state of ECN support on the endpoint and path by tracking ECN negotiation in the TCP handshake.

6.4.4.2 TFO Plugin

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

TFO [4] is an extension to speed up the opening of successive 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. 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 PATHspider plugin tests TFO negotiation as well as reconnection with SYN data.

6.4.4.3 DCSP Plugin

The DiffServ codepoints plugin for PATHspider aims to detect breakage in the Internet due to the use of a non-zero DiffServ codepoint, and also to determine if DiffServ codepoints are set on a path.

Differentiated services or DiffServ [14] 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 [9]. The DCSP PATHspider plugin tests connectivity for non-zero codepoints on SYN and reports also on the observed DCSP codepoint in the SYN ACK reply.

6.4.4.4 Evil Bit Plugin

The Evil Bit plugin for PATHspider aims to detect breakage in the Internet due to the use of reserved bit in the IP fragment offset field.

The Evil Bit refers to the unused high-order bit of the IP fragment offset field in the IP header. It was defined in [2] on the 1st of April 2003. The purpose of measuring this impairment is to determine whether this reserved bit can be reused for other protocol mechanisms in the future without connectivity impairments.

6.4.4.5 h2 Plugin

The h2 plugin for PATHspider aims to detect breakage in the Internet due to the use of HTTP/2.

HTTP/2 (originally named HTTP/2.0) is a major revision of the HTTP network protocol used in the Internet. The HTTP Upgrade mechanism is used to establish HTTP/2 starting from plain HTTP. The client starts a HTTP/1.1 connection and sends Upgrade: h2 header. If the server supports HTTP/2, it replies with HTTP 101 Switching Protocol status code. The HTTP Upgrade mechanism is used only for cleartext HTTP2 (h2c). In the case of HTTP2 over TLS (h2), the ALPN TLS protocol extension is used instead. The PATHspider plugin tests the use of h2 Upgrade and connection success.

6.4.4.6 UDP Zero Checksum Plugin

The UDPZero plugin for PATHspider aims to detect breakage in the Internet due to the use a zero-checksum field.

UDP uses a 16-bit field to store a checksum for data integrity. The UDP checksum field [16] is calculated using information from the pseudo-IP header, the UDP header, and the data is padded at the end if necessary to make a multiple of two octets. The checksum is optional when using IPv4, and if unused a UDP checksum field carrying all zeros indicates the transmitter did not compute the checksum.

7 Conclusion

This deliverable concludes the development of the Path Transparency Observatory and PATHspider active path transparency measurement tool. We now transition both of these systems to maintenance, although we may add small features to ease our day-to-day use of the platform from now to the end of the project. The master branch of the PTO repository always contains the latest version of the software running on our instance:

https://github.com/mami-project/pto3-go

The PTO forms the basis of longitudinal measurement campaigns to quantify Internet path transparency and impairments thereto; these will be reported in full in the forthcoming deliverable D1.3.

In addition, we demonstrate how the PTO provides a specific example of a generalized *observatory* design pattern for the storage and analysis of Internet measurement data.

The MAMI project has begun conversations with other projects seeking to detect impairments due to middleboxes, both for Internet Engineering reasons and for the detection of censorship. These projects will continue to use PATHspider beyond the end of MAMI and one project has indicated interest in implementing the measurement methodologies as part of a reusable C++ library that would allow for embedding on end-user mobile platforms.

References

[1] B. Aboba and E. Davies. Reflections on Internet Transparency. RFC 4924 (Informational), July 2007.

[2] S. Bellovin. The Security Flag in the IPv4 Header. RFC 3514 (Informational), Apr. 2003.

[3] E. Chen and S. Sangli. Address-Prefix-Based Outbound Route Filter for BGP-4. RFC 5292 (Proposed Standard), Aug. 2008.

[4] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413 (Experimental), Dec. 2014.

[5] R. Craven, R. Beverly, and M. Allman. A Middlebox-cooperative TCP for a Non End-to-end Internet. In *SIGCOMM*. ACM, 2014.

[6] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 1–8, Barcelona, Spain, 2013. ACM.

[7] K. Edeline, M. Kühlewind, B. Trammell, and B. Donnet. Copycat: Testing differential treatment of new transport protocols in the wild. In *Proceedings of the Applied Networking Research Workshop*, ANRW '17, pages 13–19, Prague, Czech Republic, 2017. ACM.

[8] G. Fairhurst, A. Custura, A. Venne, and T. Jones. LE codepoint: preliminary results and ongoing work in the IETF, 2017. Presentation at maprg at IETF-98.

[9] D. Grossman. New Terminology and Clarifications for Diffserv. RFC 3260 (Informational), Apr. 2002.

[10] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys Tutorials*, 16(4):2037–2064, Fourthquarter 2014.

[11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is It Still Possible to Extend TCP? In *ACM SIGCOMM IMC*. ACM, 2011.

[12] M. Kühlewind, M. Walter, I. Learmonth, and B. Trammell. Tracing internet path transparency. In *Traffic Measurement and Analysis Conference*, Vienna, Austria, 2018.

[13] I. A. Names and N. Authority. Transmission control protocol (TCP) parameters. https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml, June 2018.

[14] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), Dec. 1998. Updated by RFCs 3168, 3260.

[15] V. Paxson. Strategies for sound internet measurement. In *ACM SIGCOMM IMC*, 2004.

[16] J. Postel. User Datagram Protocol. RFC 768 (Internet Standard), Aug. 1980.

[17] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, 2012.

[18] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), Sept. 2001. Updated by RFCs 4301, 6040.

[19] D. Saucez and L. Iannone. Thoughts and recommendations from the acm sigcomm 2017 reproducibility workshop. *SIGCOMM Comput. Commun. Rev.*, 48(1):70–74, Apr. 2018.

[20] I. Swett. QUIC Deployment Experiment @ Google. Proceedings of IETF 96, July 2016.

[21] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger. Enabling internet-wide deployment of explicit congestion notification. In *Passive and Active Measurement (PAM) Conference*, Brooklyn, USA, 2015.

[22] B. Trammell, M. Kühlewind, P. De Vaere, I. R. Learmonth, and G. Fairhurst. Tracking transport-layer evolution with pathspider. In *Proceedings of the Applied Networking Research Workshop*, ANRW '17, pages 20–26, Prague, Czech Republic, 2017. ACM.