Fachhochschule Nordwestschweiz
Hochschule für Technik

CyberCaptain

Simply Analyse Data Over Time

Bachelor Thesis

BSc Informatik

Authors:
Nick Thommen, Tobias Wilcke

Supervisor and Customer:
Martin Gwerder

Expert:
Željko Blazinić

Windisch, 17.08.2018

Institut für Mobile und Verteilte Systeme

Bahnhofstrasse, 6
5210, Windisch

nick.thommen@students.fhnw.ch
tobias.wilcke@students.fhnw.ch
www.fhnw.ch

Abstract

While there are many tools and languages available to simply analyse structured data, there is nothing comparable for unstructured data. One reason for that is, that this data format is unpredictable and may change overtime. The goal of this Bachelor Thesis is to develop a tool to help simplify analysing unstructured data over time.

To achieve this objective the requirements are analysed and implemented with the SCRUM project management method.

The tool itself is developed in Python and has a modular architecture, which allows quick and simple development of new features. Each module has its own unique challenges. Most of the solutions to these challenges are based on already similar implementations from the analysis of structured data sets.

This results in the new tool named CyberCaptain, as it allows the developer to analyze the data that the internet provides. It drastically cuts down the complexity to develop new analysis and works with much less code, which drastically reduces the possibilities for bugs to occur.

Management Summary

A tool is required to analyse unstructured data over time. This tool must be very configurable, so that it can be used for a range of different tasks. This tool is now called CyberCaptain.

Each requirement is analysed for its feasibility and usability, any concerns are voiced, discussed and if necessary added to the implementation. This process includes the acquisition of new knowledge where needed, to ensure the highest quality of the product. The weekly Stand-ups allows the team to coordinate and monitor the progress, as well as discussing any important matters.

The result of this project is CyberCaptain, which is an easily configurable and extendable tool with a modular architecture. This architecture allows the easy addition of new features. The heart of CyberCaptain are the simple and compact config files which allows CyberCaptain to analyse a data set over time and generate graphs to display the new-found data.

CyberCaptain can be used to analyse any data over a period. One such example would be to monitor the vulnerability or patch frequency of a given network.

Preface

We thank the Hochschule für Technik FHNW and Mr Gwerder for the opportunity to work on this interesting project. We thank Rahel T. for the wonderful logo she designed for us and this project. Finally, we are grateful for the patience of our families and friends who put up with us during the tough times of writing this thesis. We both know we probably had little else to talk about but this project.

CYBER
CAPTAIN

Windisch, 09.08.2018
Nick Thommen & Tobias Wilcke

Table of Contents

1 Introduction

As for now there are tools to simply analyse data over time, but all of those require the data to be structured in one way or another. If these are not structured, it is necessary to employ an expert to do so or to develop a special tool or program to analyse them. This results in both higher costs of time as well as effort, while being highly prone to mistakes.

To simplify this task a new tool is developed with the goal to make analysing unstructured data as simple as it is now to analyse structured ones.

This new tool is called CyberCaptain.

1.1 Primary Goal

The primary goal is to develop a tool, which helps the customer in his future tasks, as well as to provide an architecture which allows others to extend it for their own requirements.

A modular architecture is a simple solution for this problem, but it is important to keep all stakeholders in mind. On the one hand the customer wants to have a simple solution for his current task, as well as an architecture that is easy to extend for future use cases and requirements for other users or developers.

The customer requires that the primary source of data is from the two services Censys and Shodan. These two services provide various data sets on various servers, which can be accessed via the internet. It is also possible to load data from a local source, but this is not the primary goal of this Thesis.

It must be possible to retrieve the data from these services, analyse them over time, and plot the data on graphs to visualize any changes that might occur.

1.2 Method

To achieve this goal the requirements are collected and analysed, further details for each requirement can be found in the chapter "**Error! Reference source not found.**". Not all requirements have to be implemented. A subset of the requirements is listed in the chapter "**Error! Reference source not found.**". This chapter names and defines the minimum requirements for the success of this project.

Each requirement is added to the issue list of GitLab and mapped on to predefined milestones, as defined in chapter "**Error! Reference source not found.**". This allows the team to monitor the progress on a milestone level.

The project management process is loosely based on the SCRUM method as well as the GitLab workflow. Each issue is placed in the Backlog and the authors meet for a weekly meeting to define the next steps and issues to resolve as well as to discuss any problems that occur. This allows the developers and the customer to intervene when the requirements are not understood correctly. This results in a Change Request which at the very least will define the requirement more accurate or reprioritise the issue.

1.3 Report Concept

This report discusses the result of the Bachelor Thesis "CyberCaptain". The main part guides the reader through, from the first contact with the tool CyberCaptain (the config file), and after that dives deeper into the implementation details. It will also pick out the biggest challenges for some modules, to explain what it is and how this problem is solved.

The conclusion sums up the main part and gives an insight on how to expand on this work. While the appendix contains more documents detailing different aspects of the Bachelor Thesis, as well as some guides for CyberCaptain.

2 CyberCaptain

CyberCaptain is a tool written in Python. The config file allows the configuration of a wide range of aspects. The modular architecture allows a simple and quick extension or alteration of its features.

The commands are defined in the project specific config file. It defines where to find the data and what to do with it. Each step is a config block, which is defined by each module individually.

2.1 Configuration Language Specification

The project specific config file (`ConfigFile`) starts with a few attributes defining the project (`ProjectAttr`), each one is separated by a new line (`\n`). These help to structure the project and support the developer by having everything in one place.

These attributes are comprised of the project name (`ProjectName`) and the project root (`ProjectRoot`), both are string values (`strValue`).

```
ConfigFile  := ProjectAttr \n ModuleBlock*
ProjectAttr := ProjectName \n ProjectRoot
ProjectName := projectName = strValue
ProjectRoot := projectRoot = strValue
```

After the project attributes comes any number of module blocks (`ModuleBlock`). Each block is comprised of the module name (`ModName`), which defines the module to be called. A block name (`BlockName`) identifies the individual block. In the end, there are several attributes (`Attr`). Each one of these attributes defines the behaviour and influences the result of the given module.

```
ModuleBlock := [ModName BlockName] (\n Attr)*
```

Each attribute is defined by a key (`Key`) and a value (`Value`) pair. Each module defines its own key value pairs. The key results in a string value. The value can be a string value, a number (`num`) or a list. The list consists of comma separated values.

```
Attr  := Key = Value
Key   := strValue
Value := strValue|num|, Value
```

2.1.1 Configuration Example

Given the previously defined grammar, a configuration file can look like follows.

```
projectName = "ExampleScript"
projectRoot = "/path/to/project/root"

[processing_clean CLEAN_THE_ONE]
src = "https_censys_{{currentdate}}-{{count}}.cctf"
format = "json"
keep = "ip", "domain"
removeMissingKeys = "True"
target = "https_censys_clean_{{currentdate}}-{{count}}.cctf"
```

The double quotes are optional but improves the readability, and numbers and strings can be easier distinguished.

If a Boolean is required by the module the strings "`yes`", "`true`", "`t`" and "`1`", regardless if upper or lower case, represent a Boolean value of “true”.

2.2 Interpreter

The CyberCaptain interpreter is made up of two main components:

- ConfigFile interpretation
- Validation/Run Algorithm

Loading the ConfigFile via the library ConfigObj does not provide enough information to be able to run directly. Each of the ModuleBlocks in the ConfigFile can be defined following no specific order. This makes it impossible for the run algorithm to process the ConfigFile in a strict top-down logic. First it is necessary to find all ModuleBlocks, also called `Tasks`, that depend on each other. This allows the algorithm to build `Paths` and run them afterwards.

```
###################################
# CyberCaptain Interpreter Example#
###################################

projectName = interpreterExample
projectRoot = /path/to/project

[processing_clean LOCAL_CLEAN1]
src        = example_input_data_1.ccsf
 …module attributes omitted…
target  = interpreterExample_cleaned-{{currentdate}}.cctf

[processing_filter LOCAL_FILTER1]
src      = interpreterExample_cleaned-{{currentdate}}.cctf
 …module attributes omitted…
target  = interpreterExample_filtered-{{currentdate}}.cctf

[processing_group LOCAL_GROUP1]
src      = interpreterExample_filtered-{{currentdate}}.cctf
 …module attributes omitted…
target  = interpreterExample_grouped-{{currentdate}}.cctf

[visualization_bar LOCAL_BAR1]
src      = interpreterExample_grouped-*.cctf
 …module attributes omitted…
target  = interpreterExample_plot.png

[visualization_line LOCAL_LINE1]
src      = interpreterExample_grouped-*.cctf
 …module attributes omitted…
target  = interpreterExample_plot2.png
```

2.2.1 Tasks and Paths

Related tasks are determined by first finding all the tasks which represent the final task for a path. A final task is identified by the fact that their targets will not be used as a source for another task. This is done by a lookup for the target in the list of all the defined modules. Because a few modules support wildcard patterns in their sources, the lookup is not only using string matching but also filename pattern matching (For example: “file1.cctf” ≠ “file*.cctf”).

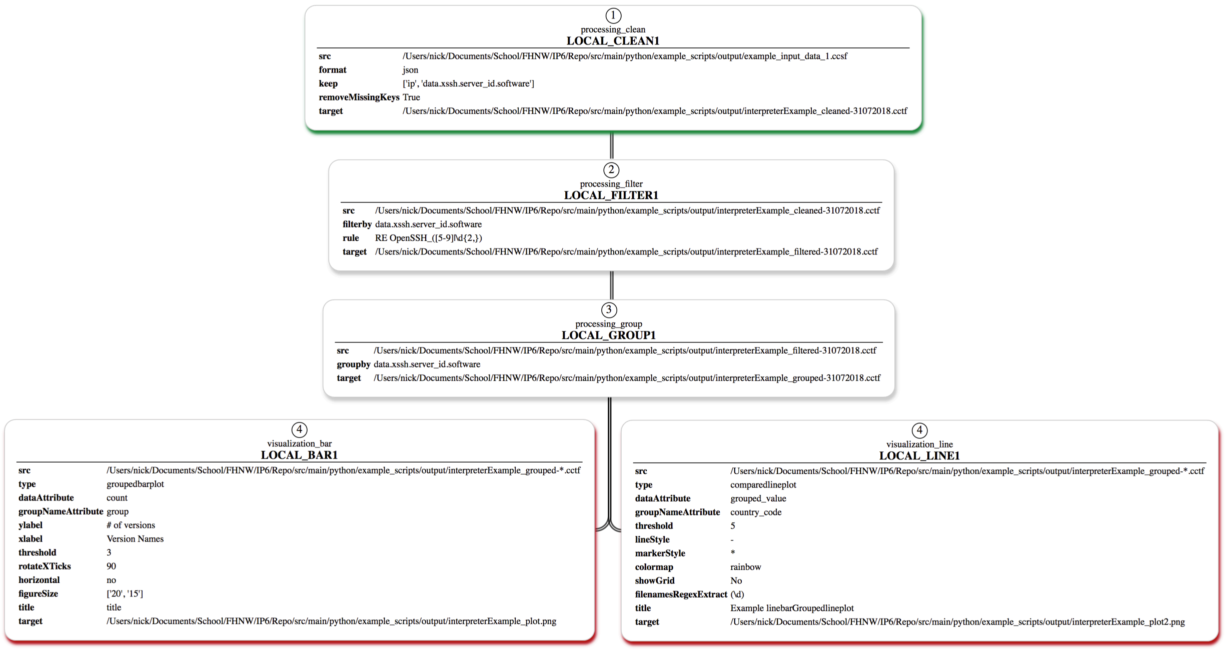
The targets “interpreterExample_plot.png” and “interpreterExample_plot2.png” are not used in any of the sources. So, the algorithm interpreting the demonstration configuration ends up with the two final tasks “`visualization_bar LOCAL_BAR1`” (LN23) and “`visualization_line LOCAL_LINE1`” (LN28).

Starting from the found final tasks, the algorithm uses their sources to determine the previous task by matching it to all the targets (again string and pattern matched due to wildcards). This bottom-up approach is repeated until it does not match to anymore targets. The last found task is the start task for the path.

This process is repeated for every single final task. Running the example through it results in two paths, these are in the order in which they are found. (Red = Final tasks / Green = Starting tasks):

```
[['visualization_bar LOCAL_BAR1', 'processing_group LOCAL_GROUP1',
'processing_filter LOCAL_FILTER1', 'processing_clean LOCAL_CLEAN1'],
['visualization_bar LOCAL_LINE1', 'processing_group LOCAL_GROUP1',
'processing_filter LOCAL_FILTER1', 'processing_clean LOCAL_CLEAN1']]
```

Both lists are almost equal, only the visualized modules differ. Even though the lists contain the same tasks, no task will be run twice. The result (target-file) of the first run will be reused in case of a rerun. This reuses as much data as possible and reduced the workload as much as possible.



```
① processing_clean
LOCAL_CLEAN1
src                /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/example_input_data_1.ccsf
format             json
keep               ['ip', 'data.xssh.server_id.software']
removeMissingKeys  True
target             /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_cleaned-31072018.cctf

② processing_filter
LOCAL_FILTER1
src       /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_cleaned-31072018.cctf
filterby  data.xssh.server_id.software
rule      RE OpenSSH_([5-9]|\d{2,})
target    /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_filtered-31072018.cctf

③ processing_group
LOCAL_GROUP1
src      /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_filtered-31072018.cctf
groupby  data.xssh.server_id.software
target   /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_grouped-31072018.cctf

④ visualization_bar
LOCAL_BAR1
src                 /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_grouped-*.cctf
type                groupedbarplot
dataAttribute       count
groupNameAttribute  group
ylabel              # of versions
xlabel              Version Names
threshold           3
rotateXTicks        90
horizontal          no
figureSize          ['20', '15']
title               title
target              /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_plot.png

④ visualization_line
LOCAL_LINE1
src                    /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_grouped-*.cctf
type                   comparedlineplot
dataAttribute          grouped_value
groupNameAttribute     country_code
threshold              5
lineStyle              -
markerStyle            *
colormap               rainbow
showGrid               No
filenamesRegexExtract  (\d)
title                  Example linebarGroupedlineplot
target                 /Users/nick/Documents/School/FHNW/IP6/Repo/src/main/python/example_scripts/output/interpreterExample_plot2.png
```

Figure 1 Paths visualized, no repeated tasks

2.2.2 Run

For the actual run, every found path is processed one after another. Each module in the reversed path (now Top-Down) is instantiated with all the configured module specific attributes (e.g. ‘`src`’, ‘`target`’, ‘`type`’, ...). Every module’s run uses the same procedure.

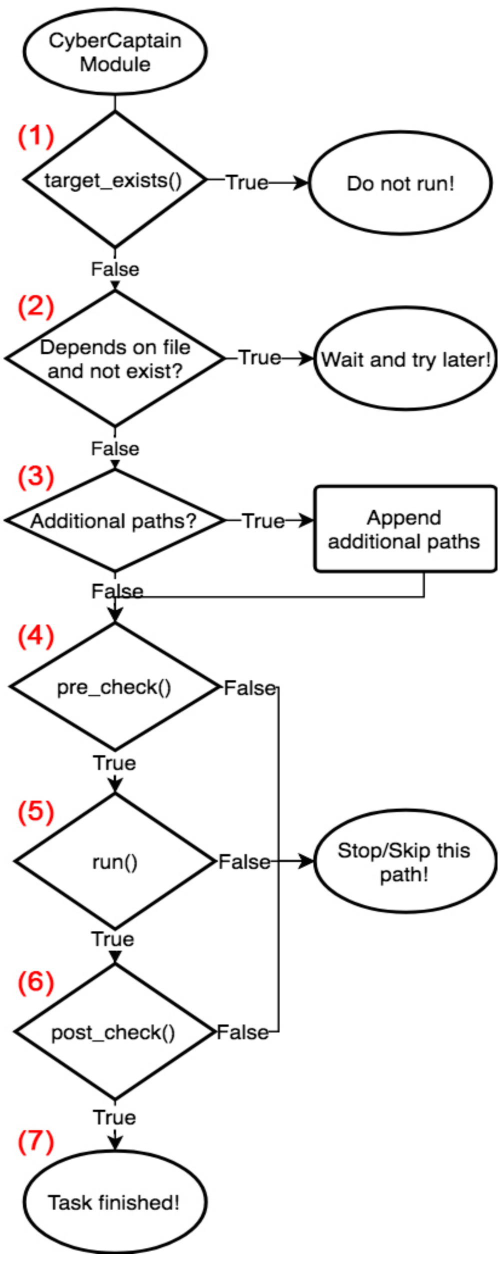


Figure 2 Run procedure sequence diagram

(1) Before every run, the instantiated module will check with its own “`target_exists()`”-Method if the defined target already was created. If that’s the case, the task was run before and it can be skipped without going further with the next task in the path.

(2) Certain tasks depend on other files as well as the previous tasks target. These files can be created by any other path in the current configuration. Such a module is able to tell the run algorithm with the “`depends_on_file()`”-Method on which file it depends. The algorithm checks if this file already exists or the path must be run again later after all the other paths in the current list are finished.

(3) If the module offers the ability to inject additional paths, they are injected and appended to the current path list with the same configuration beside the different dataset.

(4) If the module has a precondition it is checked with the “`pre_check()`”-Method. In case the check fails, the complete current path is skipped.

(5) The modules “`run()`”-Method runs the module according to the configured attributes. In case the run fails, the whole current path is skipped.

(6) The modules postcondition is checked with the “`post-check()`”-Method. In case the check fails, the rest of the current path is skipped.

(7) If the task reaches the end, it is finished and the next task in the current path can start the same procedure from the beginning. This is repeated until the path is finished.

2.3 Architecture

The architecture has two sections, the primary modular architecture and the collection of auxiliary classes called utils.

2.3.1 Modules Architecture

The modules can be compared to microservices each module has no dependency on another and controls its own data flow and storage. To ensure a consistency in development and usage interfaces abstract classes are used. This results in a three-tier hierarchy:

1. CyberCaptain_Base: Defines the minimum interface each module and module group must implement as well as provides some abstract implementations.
2. ModuleGroup_Base: Defines any group specific implementations or interfaces. As well as some more abstract implementations.
3. Modules: The concrete implementation of each module.

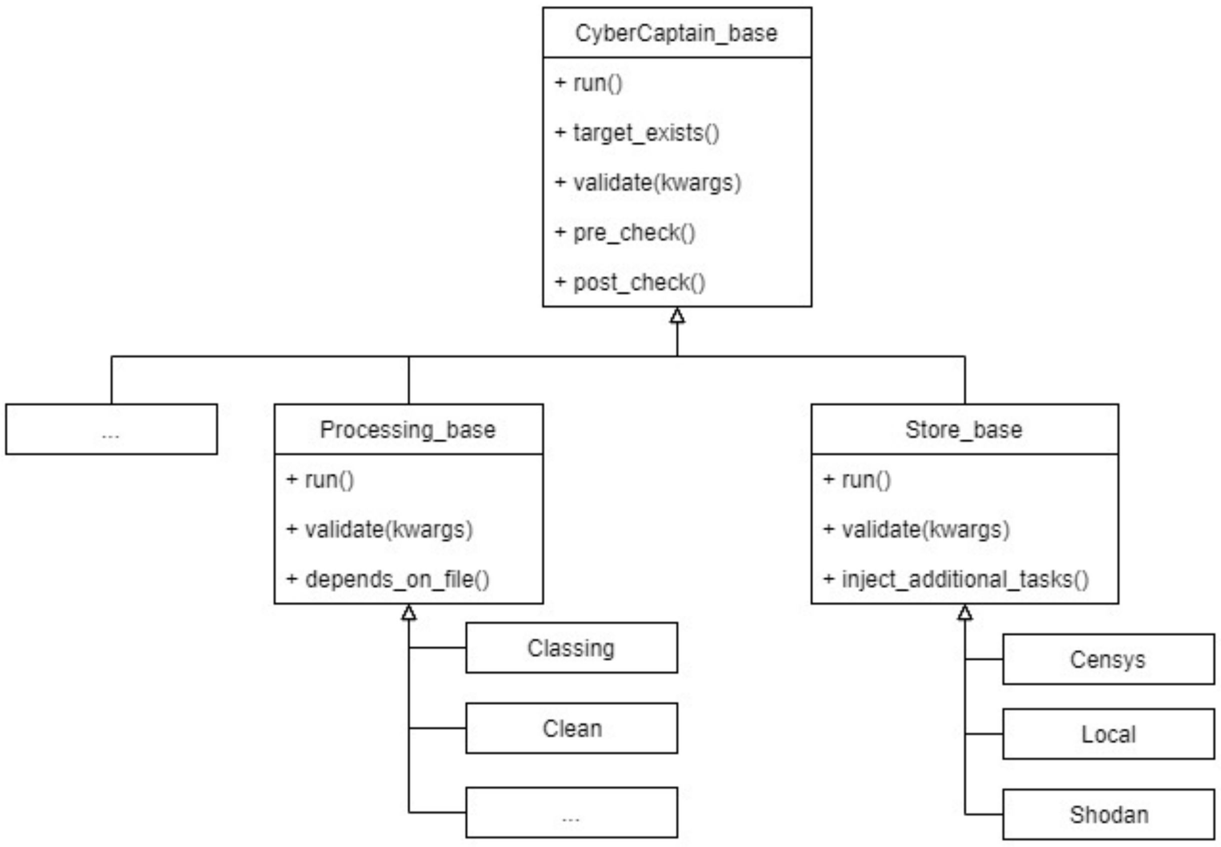


Figure 3 Small Extraction of the Architecture

Figure 3 visualizes how the hierarchy is build up and that not every method has to be implemented by the final module.

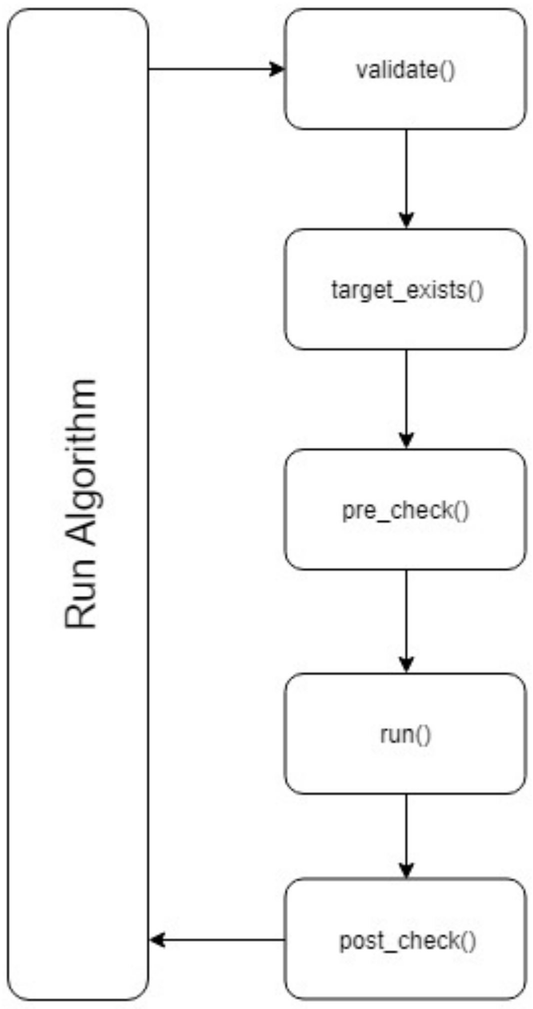


Figure 4 Module Live Cycle

The CyberCaptain_Base abstract class defines five methods. Only the `run` method must be implemented by every module. The other four provide an abstract implementation.

- `run`: The `run` method is called by the interpreter to run the central task of the module itself.

- `target_exists`: Checks if the given target already exists, if no other paths must be present as well.
Default: Checks if the target file exists.

- `validate`: The `validate` method is called during the initialisation of the module, it checks whether all the attributes from the config file are probable or even correct.
Default: Checks the existence of the source and target attributes.

- `pre_check`: Unlike the `target_exists` method, this one ensures that other prerequisites are met.
Default: Returns True.

- `post_check`: Is called after a successful completion of the `run` method. And checks if this is also possible.
Default: Returns True.

The module life cycle is straight forward visualized in Figure 4. Each step is called by the run algorithm and can influence the behaviour of every individual run, either by adding more paths, by breaking or interrupting the build because preconditions are not yet met.

ModuleGroup_Base adds little more implementtation and primarily helps to differentiate the different groups.

- Export: Exporting data to a different format.
- Processing: Morphing and changing data.
- Store: Importing data from external sources.
- Visualization: Visualizing data in different graphs.

The two groups that adjust anything on this level is the store and processing group.
The method `inject_additional_tasks` adds additional tasks if the store finds more than one new data set to be analysed. The `depends_on_file` from the processing module group, indicates that a task must be postponed because not all dependencies are present at that moment. (For example, the join module that joins two paths into one.)

The lowest hierarchy is the concrete and individual implementation of each module. This structure allows a simple extension on two fronts, more modules for a certain module group or even adding more module groups.

Either way each module must be registered in the modules.ccc file. This file contains a complete list of all installed and ready to use modules. It is in the CyberCaptain root and is separated into four sections:

- Registration: Each module needs a name which represents it in the config file and a link to the module and its class.
  ```
  <ConfigFileName> = <ModulePath>, <ClassName>
  ```
- Wildcard sources: A list of all modules which can have a wildcard in its source paths.
- Restricted targets: A list of modules which cannot be used as a source file, due to the output.
- No sources: A list of all modules that cannot have a source, because they collect their data elsewhere.

2.3.2 Utils

The primary function of the utils is simply to ensure a high code reusability and some calculations or generations following the same procedure. To help with easy use, this section is intentionally left unstructured.

The utils have a wide range of use:

- Exceptions: containing all CyberCaptain specific exceptions.
- FileHandlers: ensuring that every file is read and written the same.
- Helpers: containing several different little auxiliary methods.
- Logging: handling the entire logging.
- Key/Value Store: Also known as KV-Store it allows to save key data to a separate project specific file.

2.4 Modules

Because the modules are the backbone of CyberCaptain, the “Table 1 Supported Modules” lists all the supported modules.

Module Group	Module Name	Description
export	csv	Allows exporting data to the CSV format.
processing	classing	Allows to classify the data based on several given rules.
processing	clean	Allows to discard any unwanted data to trim the file size.
processing	country	Allows to identify the country based on the given IP.
processing	diff	Allows to see the change of the data over time.
processing	filter	Allows to filter the data sets to get rid of unwanted data sets.
processing	group	Allows to group data, either on whole attributes or with the help of a given regex.
processing	join	Allows to left join two data sets based on the given attributes.
store	censys	Allows to access data from the Censys API.
store	shodan	Allows to access data from the Shodan API.
store	local	Currently just a place holder to load local CyberCaptain Data.
visualization	bar	Allows to plot data as a bar plot.
visualization	line	Allows to plot data as a line plot.
visualization	map	Allows to plot data on a map plot.

Table 1 Supported Modules

2.4.1 Modular Challenges

Some modules present a challenge and need some research and thoughts before implementing. This results in a bit more complex implementation. This chapter discusses these modular challenges and explains how these modules work in detail.

2.4.1.1 Store Modules

Data can be loaded directly with an existing local file or via API lookups to Shodan and Censys. Censys data require the LZ4-Compressed file to be downloaded. File sizes bigger than 100GB are no rarity.

To be able to handle these sizes without taking forever and filling the memory, the download is chunked. The default chunk size of 2048 byte can be modified with a single attribute depending on the current running system.

Another challenge for both Censys and Shodan API lookups is the ability to load missing and not yet processed datasets. The APIs report a list of datasets with their timestamps or IDs.

To keep track of which datasets were processed and which were published between two runs, the KV-Store ensures to save all identifiers of processed datasets. The “inject_additional_tasks()”-Method allows to add the not processed and wanted datasets to the running paths. To prevent unforeseen issues, this makes it important to clean the KV-Store in case of the store module configuration is changed.

2.4.1.2 JSON to CSV

To further use or to store the data, it is necessary to be able to export the data. This is the function of the CSV export module. The challenge is to marry the two data formats JSON and CSV.

Both are designed to store data in a readable format. But JSON (Bray, 2018) is designed to represent Objects with a possible depth, while CSV (Shafranovich, 2018) is designed to represent a table.

To ensure that no data is lost the JSON file must be “flattened”. Meaning that every attribute regardless how deep it is nested will be brought to the same level. And then the Python to string command allows the printing of each attribute value.

2.4.1.3 Join

The join module is quite complicated because of two major reasons, memory and duration.

Efficiency is not a high priority, it cannot be ignored for this join module. Looping several times over all data sets would result in a too long of a runtime.

While for the memory it is simply impossible to store everything in memory without needing an endless number of swaps while processing.

Both problems are solved using a B-Tree from the BTrees library. This allows the join to create a B-Tree out of the right table and loop once through the left table to join them in a left join.

2.4.1.4 Clean

To only keep certain attributes of the dataset, the clean can keep or drop a set of defined attributes. A deeply nested file does not only cause the challenge to not mistakenly remove a whole tree but also brings the risk of not existing attributes in the dataset.

Missing attributes are intercepted by two options: “`ignoreMissingKeys`” or “`removeMissingKeys`”. Ignoring missing keys will keep the data row, although a key is not existing. While the option to remove missing keys will do the opposite.

2.4.1.5 Country

A raw dataset containing IP-Addresses is usually not that useful without also matching the IP to a location. The country module offers to return the ISO 3166-1 (iso.org, 2018) alpha-2 for a given IP. It is advantageous to not need a request to an online lookup service for every address.

MaxMind offers free to use GeoLite2 Country databases, which can be used with the country module. Due to the projects licencing, the module does not ship with the database directly integrated.

2.4.1.6 Visualization Modules

Big JSON-formatted datasets are not easy to interpret. The visualization modules help to display the data in three different ways: Line-, Bar- and Map-Plot.

The biggest challenge for the visualization modules is the number of configurable parameters the plotting library MatplotLib offers. Every given dataset can be different and implies a new plot. This makes it hard to find the perfect setting for every future visualization.

To still ensure a meaningful visualization, all visualization modules at this state offer plenty of settings to make every plot as dynamic configurable as possible.

2.4.1.7 Visualization of Maps

For the visualization of location-based data, a map is needed. Most of the map libraries either have the wrong licence, are not useable headless or at least require a browser to be displayed. A direct image export is usually not possible.

To tackle this challenge, GeoJSON-Maps are used. GeoJSON is a format for encoding a variety of geographic data structures (RFC 7946 (Butler, 2018)). Each country on the GeoJSON world and Europe map is referenceable by its ISO 3166-1 alpha-2/3 country code. This allows to fill each country with colour depending on the given value and colourmap and finally export it as an image.

3 Conclusion

The goal of this project is to develop a new tool which allows to analyse unstructured data over time. The tool must allow a wide range of configurable items and be easy to extend if necessary.

These goals have been achieved with CyberCaptain. This new tool is now capable of retrieving data from the services Shodan and Censys processes these and can plot these on different graphs. Every rerun of the same config file ensures that not already analysed data is rerun and only focuses on the new-found data from the sources.

The config file has a simple language and allows to control almost every aspect of each individual module.

The modular architecture coupled with the microservice idea, that hands over the data storage and flow to the induvial module, makes it very simple and quick to adapt CyberCaptain to new requirements.

3.1 Use CyberCaptain

How to use CyberCaptain is detailed in chapter “User Guide”. But what can CyberCaptain be used for?

The current society constantly collects data, some of it is not very well structured. One example is the service Shodan, it collects data from the internet and provides a service to pull them from their servers. Any data that is unstructured like this one can be analysed or pre-processed by CyberCaptain.

One such example is the European distribution of Minecraft servers and the provider that host them as shown in the figures below.

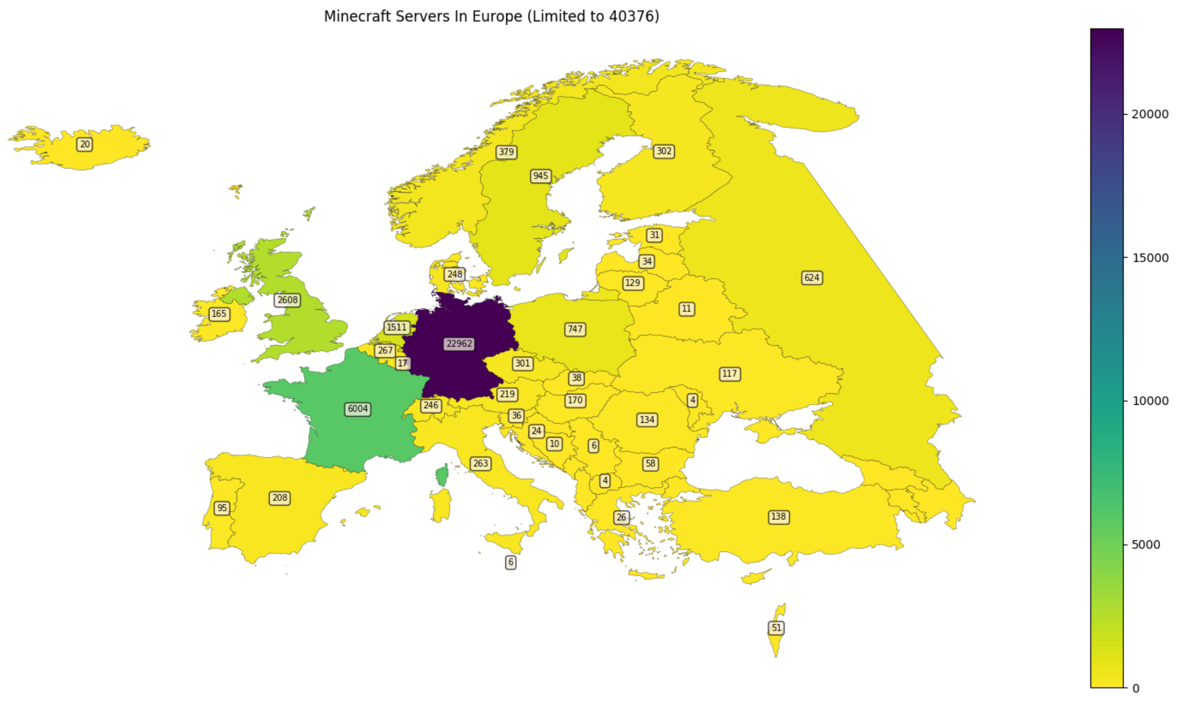


Figure 5 Minecraft servers in Europe (Limit set to 40376 server)

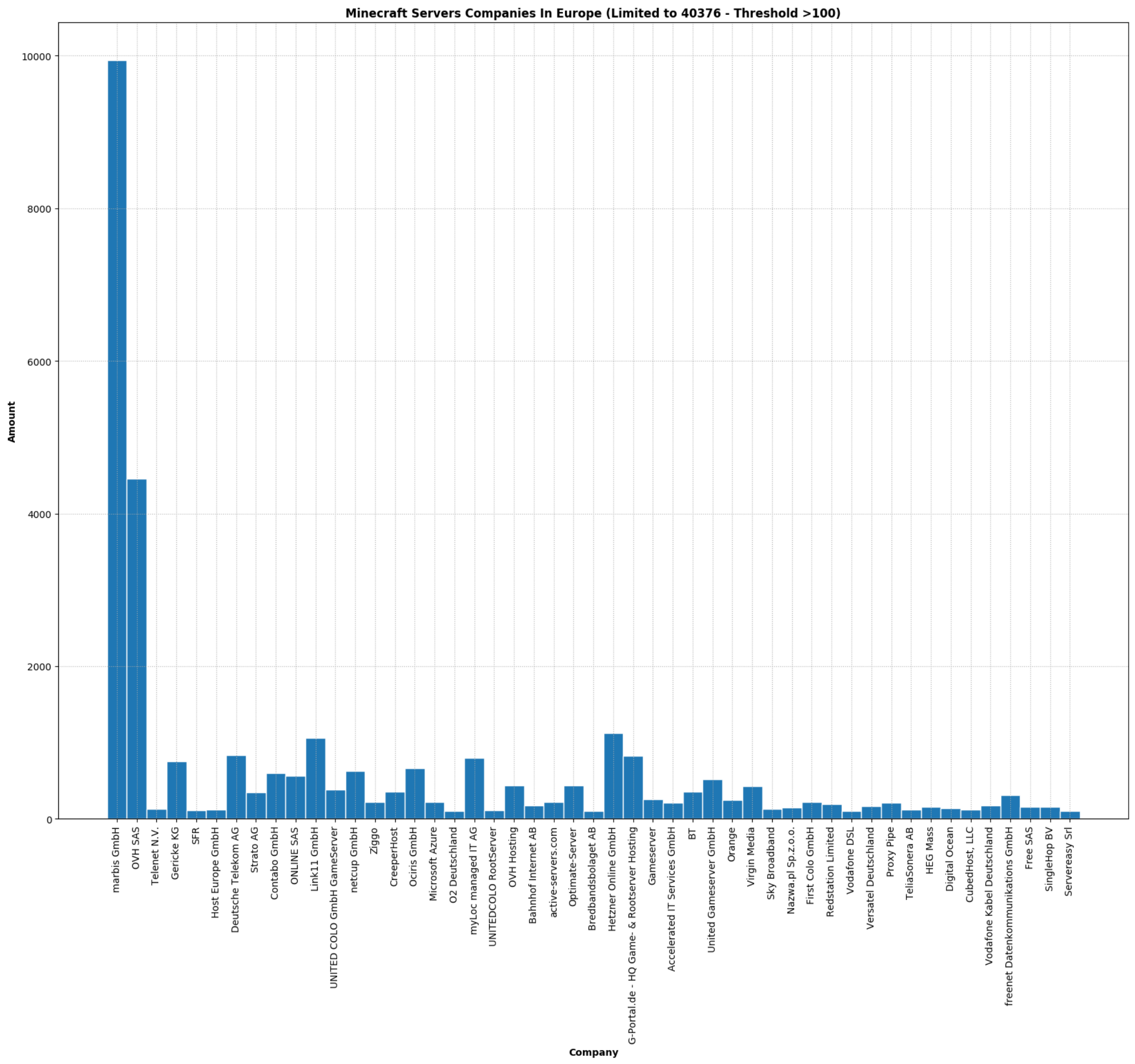


Figure 6 Minecraft server hosting companies (Threshold >100 server)

3.2 Future Road Map

CyberCaptain is far from complete, there are many more modules and features that can be added. This ranges from adding more import or export modules over the processing module group all the way to the visualisation module. It is also possible to develop a complete language around the current CyberCaptain base. This would allow the integration of arithmetic functions within the config file to calculate new columns.

4 Bibliography

Bray, T. (2018, August 2). *IETF Tools Pages*. Retrieved from tools.ietf.org/html/rfc7159

Butler, H. (2018, August 9). *IETF Tools Pages*. Retrieved from https://tools.ietf.org/html/rfc7946

iso.org. (2018, August 9). *ISO*. Retrieved from https://www.iso.org/iso-3166-country-codes.html

Shafranovich, Y. (2018, August 2). *IETF Tools Pages*. Retrieved from tools.ietf.org/html/rfc4180

5 Appendix

5.1 Table of Figures

5.2 List of Tables

5.3 Developer Guide

To develop new features for CyberCaptain it is crucial to understand the architecture behind it (See chapter Architecture). There are three places where to add new features, a new module, changing the Interpreter and the utils.

To change anything in the interpreter the corresponding chapter

Interpreter should help with any changes.
How to set up the dev environment read the README file in the repository.

5.3.1 Add new Modules

It is crucial that each module is independent from any other module.

```
class cybercaptain_base(object):
  def run(self):
  def target_exists(self):
  def validate(self, kwargs):
  def pre_check(self):
  def post_check(self):
```

The most important method to implement is `run`. It is called by the Interpreter when the module is called to work with the data.

If there are other attributes next to `src` and `target`. The validate method allows to check these before running all the modules. This is to ensure that the script can run with the given input and any run time error must be from the data.

The other methods are:

- `target_exists`: If there are other ways to check if the module has been run overwrite this method.
- `pre_check`/`post_check`: Help to make any other checks before and after running the module.

After the module at least implements the run method it must be registered in the `modules.ccc` file found in the CyberCaptain root file.

5.3.2 Utils

The Utils have no structure at all but are there to reuse as much code as possible. The most important ones are:

- `jsonFileHandler` reads and writes data from the JSON format.
- `kvStore` allows access to the project key value store file with ease.
- `logging` returns the logging instance and handles all the things around it.
- `exceptions` contain all the CyberCaptain specific exceptions that are needed.

5.4 User Guide

This guide includes how to install CyberCaptain, set up the configuration file, configuring the interpreter and executing the configuration file.

5.4.1 Installation

To use CyberCaptain, an initial installation of the latest version is needed. Under most circumstances, installing CyberCaptain is a very simple process and takes less than 5 steps to have a running instance. However, a guide for a manual installation is provided, too.

5.4.1.1 Requirements

- Python3.5 or higher
  - Verify with '`python --version`'
- pip (usually shipped with python)
  - Verify with '`pip --version`'
- git
  - Verify with '`git`'

5.4.1.2 Option 1 – One-step checkout, build & run

Running on Windows? *We recommend using Option 2 (manual installation) due to Windows specific library problems.*

When all requirements are installed, the one-step checkout can be started by running

```
sudo pip install pyb-init && pyb-init git git@gitlab.fhnw.ch:IP56/cybercaptain.git
```

This will checkout and initialise CyberCaptain in a new folder (**Attention:** checkout access to the repository is needed). After changing into the newly created folder, the virtual environment, which contains all needed python libraries, can be activated.

```
source venv/bin/activate
```

The actual build can now be started.

```
pyb
```

The build will create a new folder in '`target/dist/`' according to the CyberCaptain build version.

```
cd target/dist/cybercaptain-$VERSION
```

While being in the activated virtual environment and build folder, CyberCaptain is ready to run. A detailed running and configuration guide can be found in the following chapters.

5.4.1.3 Option 2 – Manual build & run

When all requirements are installed and a manual installation or build is needed/wanted, this guide will help with that. Additional to the requirements, the repository must be checked out manually and stored in a selected folder.

5.4.1.3.1 Installing

Starting from the projects repository folder, the continuous build tool 'PyBuilder' must be installed.

```
pip install pybuilder
```

If PyBuilder did install successfully, the project dependencies need to be installed.

```
pyb install_dependencies
```

The previous command will also run the tests and report if the build was successful.

```
------------------------------------------------------------
BUILD SUCCESSFUL
------------------------------------------------------------
Build Summary
             Project: cybercaptain
             Version: 1.0.dev0
      Base directory: /path/to/cybercaptain
        Environments:
               Tasks: install_dependencies [2799 ms]
Build finished at 2018-04-03 09:36:23
Build took 2 seconds (2811 ms)
```

Windows Speciality

The command '`pyb install_dependencies`' will fail on Windows because of the Geopanda dependency. Therefore, Geopanda must be installed manually following the instructions on the following blog:
https://geoffboeing.com/2014/09/using-geopandas-windows/

After following the instructions, the command to install the dependencies can be repeated and should run without any issues.

Running tests

The projects unit tests can be executed by:

```
pyb run_unit_tests -v
```

The command will output the test results and reposts in the folder '`target/reports/`'.

5.4.1.3.2 Deployment

To build and deploy the project, run

```
pyb publish
```

The build will create a new folder in '`target/dist/`' according to the CyberCaptain build version.

```
cd target/dist/cybercaptain-$VERSION
```

While being in the build folder, CyberCaptain is ready to be run. A detailed running and configuration guide can be found in the following chapters.

5.4.1.3.3 Generate code documentation

To automatically generate the documentation from the source code, Sphinx can be used. If new modules were added or code has been changed, please first run the following code to pick up the changes in the documentation.

```
sphinx-apidoc -f -M --implicit-namespaces -e -o docs/ src/main/python/cybercaptain/
```

If no changes were made or the previous command was executed, the documentations HTML can be generated by

```
pyb sphinx_generate_documentation
```

5.4.1.4 Verify the installation

To quickly verify the installation, the interpreter can be run without configuring any command line attributes. The following command must be executed in the previously built folder:

```
python runCybercaptain.py
```

If everything is working as expected, the interpreter should report, that no configuration file was defined.

```
  File "runCybercaptain.py", line 701, in <module>
    main(create_parser().parse_args(sys.argv[1:]))
  File "runCybercaptain.py", line 687, in main
    raise Exception("Please define the config file location [-c|--config]")
Exception: Please define the config file location [-c|--config]
```

5.4.2 Configuration

The main configuration of CyberCaptain happens through configuration files and specifications with command line attributes. This part of the guide will help to setup the configuration file and use the command line attributes.

5.4.2.1 Config File

A basic config file consists of comments, required top level attributes, task definitions.

5.4.2.1.1 Comments

Comments can be added by adding a number sign (#) in front of the line.

```
# This is a comment
```

5.4.2.1.2 Required Top Level Attributes

As of now, there are two required attributes:

Attribute	Functionality
projectName	Defines the name for the current project. Used for modules with project specific functionality (e.g. KV-Store).
projectRoot	Defines the root folder for the project where all project specific files (including files produced from tasks) will be stored. The project root counts as the absolute path to every following defined source or target.

Table 2 Required Top Level Attributes

```
projectName = thisIsAnExample
projectRoot = /absolute/path/to/project
```

5.4.2.1.3 Module Blocks

Following the required top-level attributes, the module blocks and tasks can be defined. Each module block has a top part, defining the module and unique task/block name. And a bottom part defining the module specific attributes.

[moduleName blockName]
src = xyz.ccsf
attr1 = str1
attr2 = el1, el2
attr3 = 100
target = xyz2.cctf
	moduleName defining one of many modules offered by CyberCaptain; blockName defining a unique identifier for a specific block (freely selectable)

	The module specific attributes are defined below the top-part (separated by square brackets)

Table 3 Module Blocks Explained

For a list of all supported moduleNames consider “Table 1 Supported Modules”. For all module specific configuration attributes, please consider the web documentation published on the FHNW pages[1] or generate the code and module documentation manually described on chapter “Generate code documentation”.

Remember: moduleNames consist of the ModuleGroup and ModuleName (combined by an underscore in lowercase – e.g. ‘`processing_clean`’).

The goal of CyberCaptain and the configuration file is, to combine multiple module blocks and tie them together by reusing a previous module blocks target as the source for the new module block. This allows to process one module block after another and work on the previous modules output data. For a more complicated setup, multiple module blocks can be defined which do not work on the same files. This implies multiple independent paths to the interpreter.

5.4.2.1.4 Placeholders

To ensure that the data can be analysed over time and not lose any data during the process. It is possible to put placeholders anywhere within the configuration file. In the use case to analyse over time it’s recommended to make use of them on the src and targets. Each place-holder is surrounded by two curly brackets ‘{{…}}‘ and has a defined key word.

These keywords can be defined by the developer by passing them as arguments with the flag ‘-cp’ followed by the keyword and the value with which it is replaced, for example

```
-cp count=1
```

There are currently two exceptions. The key word ‘currentdate’ and ‘currentdatetime’ are the only ones that do not need the ‘-cp’ flag and replaces the placeholder with the current date or datetime.

[1] https://ip56.pages.fhnw.ch/cybercaptain/

5.4.2.1.5 Example

A finished two paths setup, with one path producing an independent file which will be joined to the second path later, can look as follows:

```
# **************************************************************
# Example Join Module From Other Task Script Config
# --------------------------------------------------------------
# Joins a file which is generated by another task to the given src
# --------------------------------------------------------------
# How to use: Change projectRoot & copy example_input_grouped
# files to the configured projectRoot.
# **************************************************************
projectName = joinFromDifferentTaskExample
projectRoot = /path/to/project/root

# Processing Join Depends On A File Generated By "cleanTaskName2"
[processing_clean cleanTaskName1]
src = example_input_data.ccsf
format = json
keep = "data.xssh.server_id.raw", "ip"
ignoreMissingKeys = True
target = joinFromDifferentTaskExample_cleaned_1.cctf

[processing_join joinTaskName]
src = joinFromDifferentTaskExample_cleaned_1.cctf
joinwith =  joinFromDifferentTaskExample_cleaned_2.cctf
left-joinon = "ip"
right-joinon = "ip"
target = joinFromDifferentTaskExample_joined.cctf

# Needs to be done first before join can be executed on path 1 - due to order
[processing_clean cleanTaskName2]
src = example_input_data.ccsf
format = json
keep = "data.xssh.server_id.raw", "ip"
ignoreMissingKeys = True
target = joinFromDifferentTaskExample_cleaned_2.cctf
```

More script configuration examples can be found in the build source folder `‘example_scripts’.`

5.4.2.2 Interpreter CLI

Command line attributes are used to configure the core interpreter for CyberCaptain.

Command		Functionality
-c (*--config)*	☑	Defines the path to the wanted running configuration file. ***Example****: python run.py –c /path/to/config.ccs*
-mc *--modulesconfig)*	☐	Defines a custom modules config file. Defaults to ‘modules.ccc’. ***Example****: python run.py –c /conf.ccs –mc /path/mod.ccc*
-v *(--validate)*	☐	Defines that the configuration file should only be validated without running. ***Example****: python run.py –c /conf.ccs -v*
-d *(--debug)*	☐	Defines that the logging output should be set to debug level and redirect the output to STDOUT. ***Example****: python run.py –c /conf.ccs –v -d*
-cp *(--customplaceholders)*	☐	Defines and fills in the custom placeholders in the configuration file. ***Example****: python run.py –c /conf.ccs –d –cp count=1 –cp k=v*
--overwritechecksum	☐	Overwrite the given configuration file checksum. ***Example****: python run.py –c /conf.ccs –d --overwritechecksum*
--ignorechecksum	☐	Ignore the configuration file checksum check. ***Example****: python run.py –c /conf.ccs –d --ignorechecksum*
-pv *(--pathvisualize)*	☐	Defines that the configuration file should only output the path visualization without running. ***Example****: python run.py –c /conf.ccs –d -pv*

Table 4 Interpreter CLI commands

* ☑ *= Required /* ☐ *= Not Required*

Only the configuration file location attribute is required for CyberCaptain. Every other attribute depends on the use case and file.

5.4.3 Run

5.4.3.1 Checksum Check

Before every run, CyberCaptain generates a checksum over the whole configuration file. If the configuration file changes, the interpreter will stop and warn the user of a changed configuration file. This ensures that the user is aware of possible unforeseen issues due to modules writing to the KV-Store.

The checksum check can be very annoying, especially during development. Therefore the "--overwritechecksum" or "--ignorechecksum" flag can be used. Most importantly if the configuration file makes use of store modules with usage of also getting missing datasets, it is recommended to fully delete the generated KV-Store file and rerun (named the same as projectName).

5.4.3.2 CLI Mode

The most basic running mode is the CLI mode. If everything went smoothly on the installation step, the CLI-Mode can be directly used. While being in the build folder the CLI mode can be used with:

```
python runCybercaptain.py –c /path/to/config
```

To configure the interpreter, all command line arguments described on chapter "

Interpreter CLI", can be added after the required arguments.

```
python runCybercaptain.py -c /path/to/config -d -v -cp count=1
```

5.4.3.3 Graphical Mode

CyberCaptain also allows to run in a graphical mode. However, the graphical mode requires a framework python version[2], wxPython[3] and Gooey[4] installed.

```
pythonw runCybercaptainGui.py
```

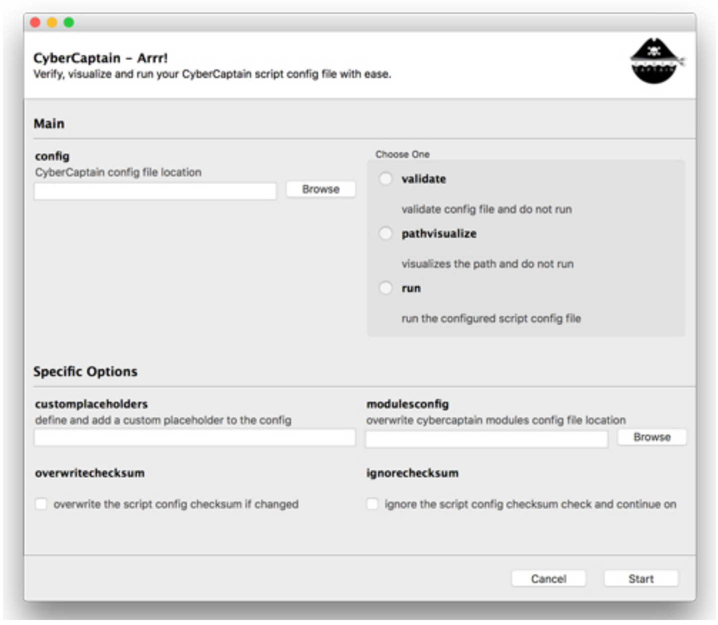

CyberCaptain – Arrr!
Verify, visualize and run your CyberCaptain script config file with ease.
Main
config
CyberCaptain config file location
Browse
Choose One
validate
validate config file and do not run
pathvisualize
visualizes the path and do not run
run
run the configured script config file
Specific Options
customplaceholders
define and add a custom placeholder to the config
modulesconfig
overwrite cybercaptain modules config file location
Browse
overwritechecksum
overwrite the script config checksum if changed
ignorechecksum
ignore the script config checksum check and continue on
Cancel
Start

Figure 7 CyberCaptain Graphical Mode

5.4.3.4 Debugging

Especially during development, debugging can be of importance. CyberCaptain does write a Log-File into the ProjectRoot at any time. For convenience purposes, log output can be redirected to the STDOUT with setting the flag "**-d**" on the command line. This also sets the verbosity level to DEBUG.

To verify a configuration file and develop it, a combination of "**-v**" and checksum flags is recommended. Paths can be verified with the included PathVisualizer (**-pv**).

[2] For macOS an anaconda python installation is recommended

[3] https://wxpython.org/

[4] https://github.com/chriskiehl/Gooey