POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

CPA/Template attacks output viewer

Project for the Embedded Systems course

Computer Science and Engineering

Leonardo Gargani, 10569221
Olivier Wartique, 10844629

Supervisor:
Giuseppe Massari

Co-supervisor:
Michele Piccoli

Academic year:
2021-2022

Abstract: This report summarizes the implementation of a GUI tool developed in Python to visualize the large amount of data generated by a CPA/template attack. Our application proposes a solution to the poor performance of conventional plotting tools when handling such amount of data. We take advantage of the static nature of values to be plotted. Therefore an initial preprocessing phase allows us to improve the overall performance and obtain a better user experience.

1. Introduction

This report summarizes the project carried out for the Embedded Systems course. The goal was to create a GUI application in order to visualize the large amount of data generated after an attempt of CPA/template attack. Conventional tools for plotting are performing poorly when handling such an amount of data (about 10 million sample points per plot). Because of it we focused our efforts to create a user friendly GUI that is capable of zooming and scrolling efficiently even in such difficult scenarios.

The report is divided in three sections. First, we discuss the implementation: what libraries have been used, what kind of input files are required, the preprocessing steps and the implemented features. Then, a section is dedicated to the issues encountered while creating the application and how they have been overcomed. Finally, last section summarizes the results of tests on time performance and memory consumption. The conclusion will then give the reader some suggestions about possible improvements and future work.

2. Implementation

2.1. Libraries used

The implementation is mainly build around existing libraries. Data handling and mathematical operations (such as peak detection) are carried out with `NumPy`, a fundamental package for scientific computing in Python. The GUI is built with the help of `PyQt5`, which is the Python equivalent of the C++ Qt framework. Finally, the plots are realised using `PyQtGraph`, a data visualisation package based on `PyQt`.

2.2. Required input files

The input files are stored in the `data/input/csv` folder. Each file is representing one of the 16 total bytes. They are saved in the `.csv` format, where the rows are time instants (up to tens of thousands rows) and the columns are correlation values generated by the attack (256 columns). The values are in the range $[-1, 1]$, and they are used to determine the success or failure of an attack.

2.3. Pre-processing steps

Before running the main program for the first time, it is necessary to execute two preprocessing steps in this specific order:

1. `csv_to_npy_conversion.py`, this script converts the `.csv` files to `.npy`. The `.npy` format is the standard binary file format in `NumPy` to save a single arbitrary array on disk.
2. `peak_detection.py`, this script is executed once the `.npy` files are generated. The aim of this step is to detect the peak value of each byte. The number of the row of the peak value, the peak value itself, and the number of its column (in the corresponding `.csv` file) are written into a dedicated file (`data/output/csv/peaks.csv`) which will then be used by the main program.

2.4. Implemented features

2.4.1 Mouse hover

The movement of the mouse cursor is generating signals (emitted by built-in slots of `PyQt` and `PyQtGraph`). These signals are connected to a custom callback function we have written. The position of the mouse is caught by such a callback and mapped from pixels coordinates to plot coordinates. These values are then shown on the graph window to help the user in visualizing data. Moreover, a crosshair symbol is added to the plot area to highlight the position of the cursor.

2.4.2 Signal generation

If the user wants to analyze a specific value, he has the possibility to click on that particular point. This will automatically emit a custom signal (named `message`) with 3 arguments: `x, y, line`. `x` is an `int` corresponding to the sample number, `y` is a `float` corresponding to the correlation value, and `line` is a `str` with the number of the line. The user can use this signal and implement his own callback function with some desired additional features. Currently, as a proof of concept, this signal is caught and the informations are simply logged to the console.

2.4.3 Zoom features

The user can also zoom using different commands. Scrolling up and down corresponds to zooming-in and zooming-out the whole plot centered on the cursor position. Holding the right click pressed and dragging horizontally will zoom on the x axis, while the same behaviour on the y axis is achieved by dragging vertically. A final possibility to zoom is to make a single right click anywhere on the plot and select `Mouse Mode > 1 button` from the contextual menu. Then, drawing a rectangular region will perform a zoom to fit that region of interest to the window.

2.5. Responsive GUI

An effort was made to realize the GUI fully responsive, so that every window is resizable while mantaining all its widgets perfectly usable. In addition to that, it is worth mentioning that the layout of every window is loaded from its own `.ui` file. The beneifts of this decision are that we have as much separation as possible between the logic of the program and its graphical aspects, and the layout can be easily modified using an IDE like `QtCreator` (which was the one we used).

2.6. Nice to have improvements

Some safety measures have been implemented to guarantee the right course of action of the application. Our implementation makes an extensive use of files I/O, both for reading and writing data. If, by any chance, one of the file is not placed in the correct directory or not present at all, an error message will be printed to the console and the script will terminate. Also, some correlation values in the files might be corrupted (i.e., NaN values). The preprocessing scripts check for that and replace them by null values (i.e, zero). An error will be also thrown if those two scripts are not run or run in the wrong order.

3. Faced problems and solutions

During the development of the application, we had to face up some problems, mainly due to the size of the dataset to handle. We think it can be useful to share here some insights about our decision-making process as the developement was moving forward, and the solutions we came up with.

3.1. The starting point

Starting from the very beginning, our goal is plotting the content of a `.csv` file, while maintaing the possibility of moving around inside the plot (i.e., zoom-in, zoom-out, change the X-range).

Our initial approach was plotting all the file at once. The problem that immediately arised was that handling over 7.5 million points (every `.csv` file has about 35 thousand rows and 256 columns) was definitely too resource-demanding. That led to an evidently unusable GUI due to its poor performance, not only when moving around, but also for the loading time of the plot itself.

Therefore, we started experimenting looking for some possible improvements to load data into memory in an efficient way using Python.

3.2. Choosing the fastest file format

A first available option we found was converting the `.csv` file into a `.hdf5` one. `HDF5` is a file format built for fast I/O processing and storage, whose potential can be fully exploited through a handy Python library.
However, the result we obtained was still very poor and the `HDF5` option had to be discarded.

Since we handled all the data as `NumPy` arrays, we also came across the `.npy` file format. It is a simple format for saving `NumPy` arrays, in an optimized way, with no loss of information. After converting the `.csv` file into a `.npy` one, we plotted it in the same way as before. The loading time was significantly lower and the usability of the GUI improved, but still far from an acceptable result.

3.3. Downsampling

Having understood that `.npy` was a far better choice over `.csv` and `.hdf5`, we decided to keep that conversion step and start tweaking around to gain as much performance as possible, this time in the `PyQtGraph` plotting process.
A special tweak worth mentioning is the way we subsample data in order to be displayed in the plot. `PyQtGraph` provides three downsampling modes: subsample, mean, and peak.
Despite being the slowest one, we picked the last one ("peak") since it downsamples by drawing a saw wave that follows the min and max of the original data. This is the only way to make sure that the peak of our plot does not get lost while displaying the points.

3.4. Plotting one curve at a time

Back to the fact that the GUI responsiveness was far from being acceptable, we came to the conclusion that showing all the 7.5 M points at the same time was not feasible in any way.
At this point we tried to change our approach. Here was (and currently is) our new one.
Since every file contains 256 rows, and each row corresponds to a curve in the plot, we can preprocess every file and identify for it which is the peak and what row it shows up in. Then, save the results in a new file. When the user opens a plot, now the program reads the content of such a file to display by default only the curve

with the peak. In addition, each one of the other curves can be shown/hidden in the plot by using a dedicated checkbox.

This new approach leads to a very low loading time and to a GUI which is not only usable, but which has high performance in every zoom/scroll operation.

Despite the high performance increase intrinsic to the new way of displaying data, we still decided to keep the conversion step from `.csv` to `.npy`. In fact, the same file stored as `.npy` takes more than 40% less space on disk than if stored as `.csv`. As we will cover in more detail in the next section, this leads to both faster loading times and lower memory usage at runtime.

4. Experimental results

Here are some useful measurements about execution time and memory usage of the program.

4.1. Testing conditions

All the tests have been performed on a machine with an i7-7500U CPU (dual-core @ 2.70 GHz) and 8 GB of RAM. Keep in mind that the execution time will vary from machine to machine.
Since at the moment of writing the hardware above can be considered mid-low end, you can likely expect better results when running the scripts.

The 16 `.csv` input files have each one with 256 columns and about 30k rows.

4.2. Conversion from .csv to .npy

The preprocessing step of converting the 16 files to the `.npy` format is performed by the `csv_to_npy_converter.py` script.
It runs for 2m00s while using 1.1 GB of memory.

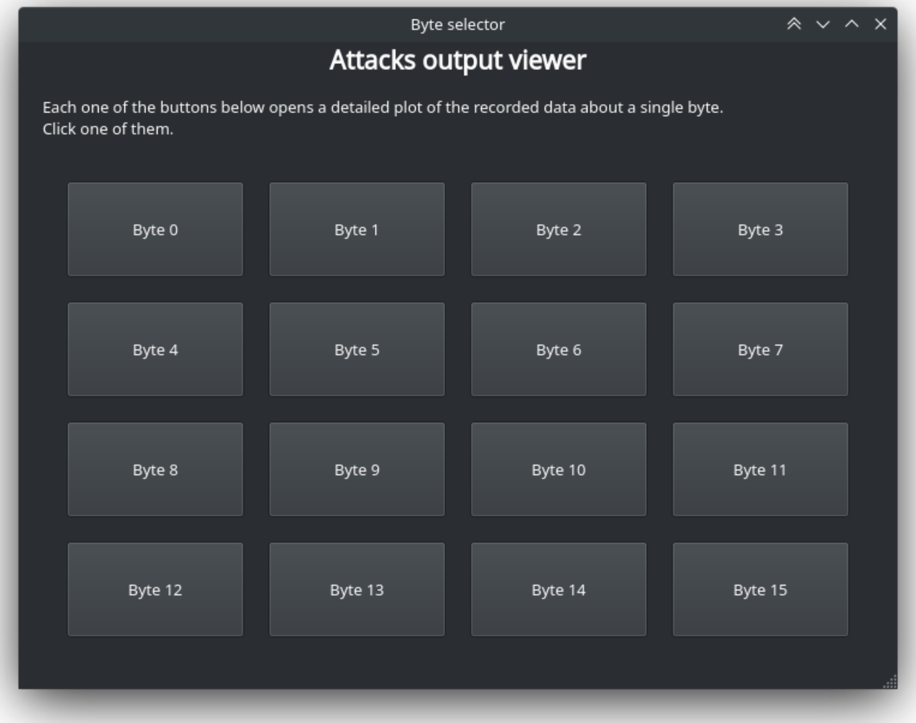
4.3. Peak detections

The preprocessing step of detecting the peak value of each one of the files is performed by the `peak_detection.py` script.
When the corresponding flag is set, in addition to detecting the peaks it also generates and saves the plots of all the files. In such a case the total execution time is 1m55s, with a memory consumption of 550 MB.
However, by default the plots are not generated and this way the execution time is extremely reduced down to just 1s.

4.4. Main program

For the main program (`main.py`), we recorded the change in memory consumption by varying the number of open plots and displayed curves.
In particular, the obtained results in memory consumption are the following:

- the main window uses 125 MB;
- every other window uses:
  - 65 MB for loading the .npy file (which is the size on disk of the file);
  - 5 MB for the creation of the window;
  - 1 MB for every displayed curve.

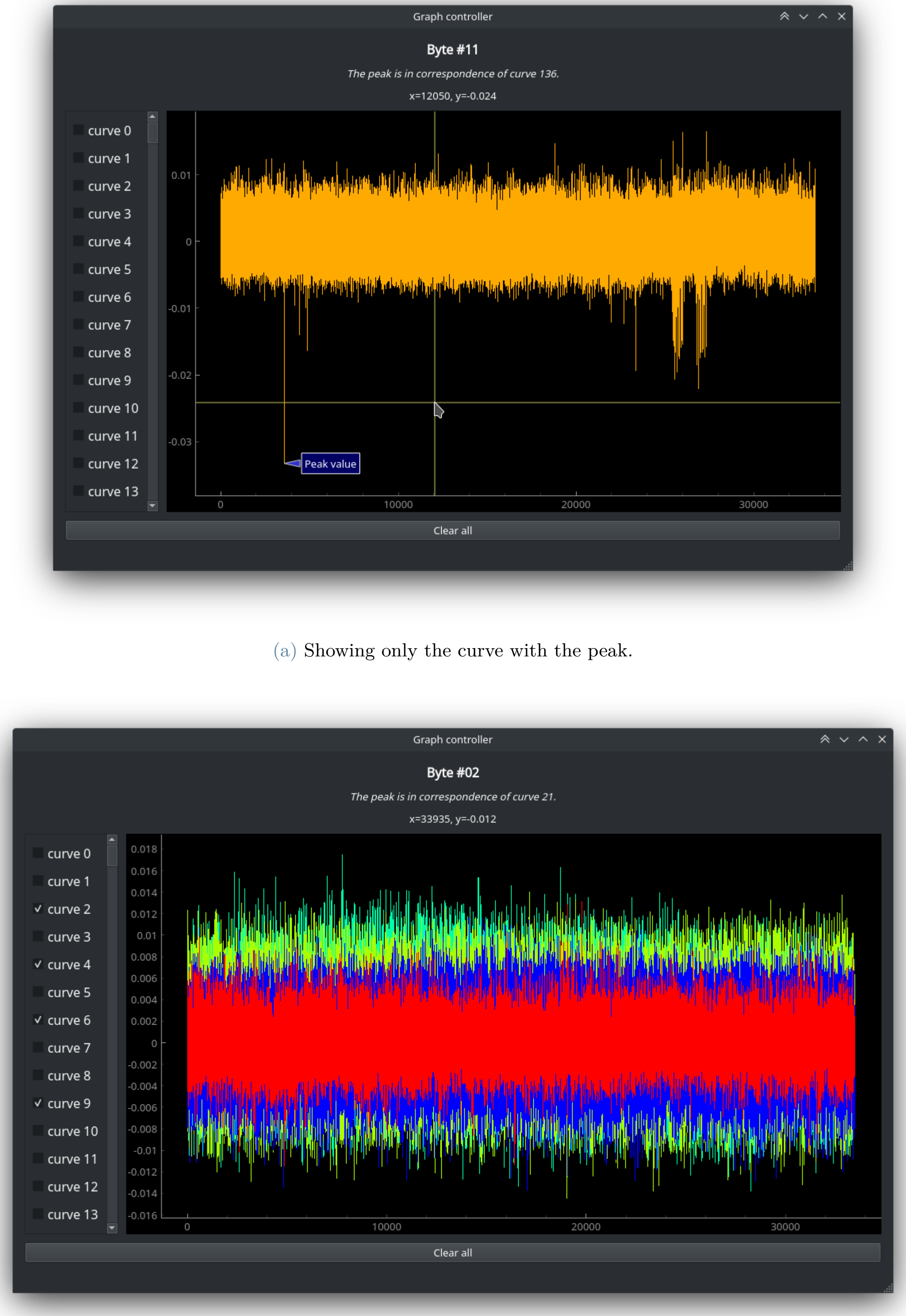


Byte selector

Attacks output viewer

Each one of the buttons below opens a detailed plot of the recorded data about a single byte.
Click one of them.

Figure 1: Main window of the program.



Graph controller

Byte #11

The peak is in correspondence of curve 136.

x=12050, y=-0.024

☐ curve 0
☐ curve 1
☐ curve 2
☐ curve 3
☐ curve 4
☐ curve 5
☐ curve 6
☐ curve 7
☐ curve 8
☐ curve 9
☐ curve 10
☐ curve 11
☐ curve 12
☐ curve 13

(a) Showing only the curve with the peak.

Graph controller

Byte #02

The peak is in correspondence of curve 21.

x=33935, y=-0.012

☐ curve 0
☐ curve 1
☑ curve 2
☐ curve 3
☑ curve 4
☐ curve 5
☑ curve 6
☐ curve 7
☐ curve 8
☑ curve 9
☐ curve 10
☐ curve 11
☐ curve 12
☐ curve 13

(b) Showing multiple curves.

Figure 2: Plot of one file.

5. Conclusions

...